Semantic Program Alignment for Equivalence Checking

Berkeley Churchill
Stanford University
USA
berkeley@cs.stanford.edu

Oded Padon
Stanford University
USA
padon@cs.stanford.edu

Rahul Sharma
Microsoft Research
India
rahsha@microsoft.com

Alex Aiken
Stanford University
USA
aiken@cs.stanford.edu

Abstract

We introduce a robust semantics-driven technique for program equivalence checking. Given two functions we find a *trace alignment* over a set of concrete executions of both programs and construct a product program particularly amenable to checking equivalence.

We demonstrate that our algorithm is applicable to challenging equivalence problems beyond the scope of existing techniques. For example, we verify the correctness of the hand-optimized vector implementation of `strlen` that ships as part of the GNU C Library, as well as the correctness of vectorization optimizations for 56 benchmarks derived from the Test Suite for Vectorizing Compilers.

CCS Concepts • **Software and its engineering → Formal software verification**; *Compilers*.

Keywords verification, equivalence checking

ACM Reference Format:
Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314596

1 Introduction

Equivalence checking, the problem of formally proving that two functions or programs are semantically equivalent, is a long-standing and important problem; applications include verification of compiler correctness [26], superoptimization [3, 6], program synthesis [29], and verifying the correctness of code refactoring [28].

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314596

```
f() {
  while(*)
    A;
  return a;
}
```
(a) Function f

```
g() {
  while(*)
    B;
  return b;
}
```
(b) Function g

```
X() {
  while(*)
    A;
  while(*)
    B;
  assert(a == b);
}
```
(c) Naive Composition

```
Y() {
  while(*)  {
    assert(Inv);
    A;
    B;      }
  assert(a == b);
}
```
(d) Syntactic Composition

Figure 1. Two functions and two product programs. A and B are basic blocks and share no variables.

Program equivalence checking is commonly performed in two stages: the first stage is to construct a *product program* for the two programs by *aligning* them, and the second is proving a safety property, or *invariant*, of the resulting program [4, 40]. There is a trade-off between the effort put into each stage. For example, consider the functions f and g in Figures 1a and 1b, where each function iterates over some loop-free basic block. A simple product program for f and g is shown in Figure 1c, where one program is run after the other; one may check the invariant that the outputs of f and g are equal. However, this alignment provides no help in checking equivalence, which requires completely summarizing the loops of f and g.

In some cases, a better alignment is to pair iterations of the loops of f and g, as pictured in Figure 1d. This alignment sometimes facilitates an easier, inductive proof of equivalence in which corresponding loop-free code fragments are shown to preserve an invariant Inv for each loop iteration [32]. However, such syntactically constructed alignments only work in simple cases where the loops of f and g execute for the same number of iterations. This is not the case for several common loop optimizations that alter syntactic structure, such as vectorization, loop unrolling, and loop

peeling. What is needed is a semantically guided alignment that relates the two programs in a way that is designed to make the final proof of equivalence as simple as possible.

We introduce a novel and robust technique for constructing product programs driven by semantics, rather than syntax, that extends equivalence checking to real-world benchmarks that are beyond the reach of prior work. Given two functions f and g along with test cases provided by the user, we build a *trace alignment*, which is a pairing of states in execution traces of f and g for each user-provided test case. Constructing the trace alignment is guided by the selection of a weak invariant, called an *alignment predicate*, that identifies pairs of machine states that should be aligned. Only once we have identified a trace alignment based on semantic properties of the programs do we lift the alignment back to the program syntax, construct a product program, and learn invariants that we attempt to prove. This approach to constructing the product program, wherein we first solve the problem of semantically aligning the traces, is the novel contribution that allows us to verify equivalence where techniques described in prior work are inapplicable.

Our goal is to perform black-box verification of optimizations performed by compilers, superoptimizers or by hand without any foreknowledge of the transformations applied or toolchains used. Therefore we evaluate our technique directly on `x86-64` assembly. Given two functions, our technique utilizes a set of user-provided test cases to guess a set of candidate alignment predicates. For each alignment predicate, we infer the trace alignment and attempt to construct a *program alignment automaton* (PAA) that specifies a product program. We again use the test cases to learn the invariants of the PAA. Finally, we use an SMT solver to check proof obligations that establish the equivalence of the functions.

We demonstrate the ability to verify several types of loop optimizations, including loop unrolling, loop peeling, vectorization, software pipelining, strength reduction, loop-invariant code motion, register allocation and loop inversion, among others. We evaluate our technique on 56 realistic loop benchmarks where compilers (`gcc-4.9.2` and `clang-3.4`) automatically perform a number of these optimizations, at least including vectorization. We further apply our technique to verify the correctness of the hand-vectorized C implementation of the `strlen` function that ships with GNU C Library (`glibc`, version $\geq$ 2.10.1), and also show that our method can verify benchmarks used to evaluate other state-of-the-art equivalence checkers.

Our contributions are:

- A novel and robust approach for semantics-driven construction of product programs using alignment predicates and trace alignments.
- A set of 56 realistic `x86-64` benchmarks for evaluating equivalence checking techniques on optimizations that alter control flow, such as loop unrolling, loop peeling and vectorization.
- The first fully automatic black-box algorithm for proving the correctness of vectorization optimizations as performed by modern compilers on `x86-64`.
- A demonstration of a useful, real-world application of our equivalence checking technology to verify the correctness of a handwritten vectorized implementation of the `strlen` function shipped in `glibc`.

The rest of the paper is structured as follows. First we introduce our running example (Section 2) before presenting the formalisms used in our work (Section 3). Then follows our equivalence checking procedure (Section 4) and evaluation (Section 5). We then present related work (Section 6) and conclude (Section 7).

2 Example

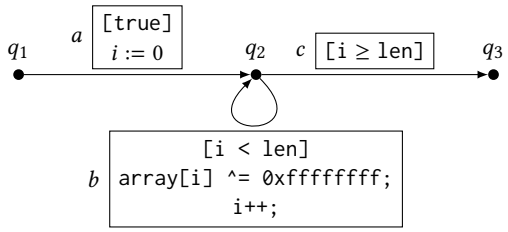
Consider the pair of C programs in Figure 2. Each function is represented as a control flow graph (CFG); the nodes are program points, and the edges are basic blocks along with a guard predicate. These functions take as input two parameters: `array`, which points to an array of 32-bit integers, and `len`, which specifies the length of the array. Both functions flip the bits of each array element. Function f (Figure 2a) iterates over each element in the array with a counter variable `i`, while g (Figure 2b) illustrates a simple way to vectorize this code using a 64-bit operation. In g, the loop body c' (lines 8-10) flips the bits of two array elements. Before the loop, there are two possibilities. If `len` is odd, block a' (lines 3-5) executes and flips the bits of the first array element. Otherwise, block b' executes and leaves the array untouched.

This example is representative of a number of challenges that naturally arise in the presence of loop optimizations. For example, if the loop in g iterates n times, then the loop in f iterates for either $2n$ or $2n + 1$ iterations, depending on the parity of `len`. Previous equivalence checking techniques handle situations where the relationship between the number of iterations of f and g is static (e.g. if g iterates n iterations then f iterates $2n$ iterations for all inputs). As a result, prior automated equivalence checking approaches [3, 7, 13, 14, 27, 32] fail on this example.

Our technique requires as input a set of test cases $\tau_1, \ldots, \tau_n$. The test cases may, for example, be generated randomly or by bounded model checking. We execute f and g on each test case to obtain traces. Figure 4 shows traces for each program with `array` initialized to address `0x100000` and `len=5`.

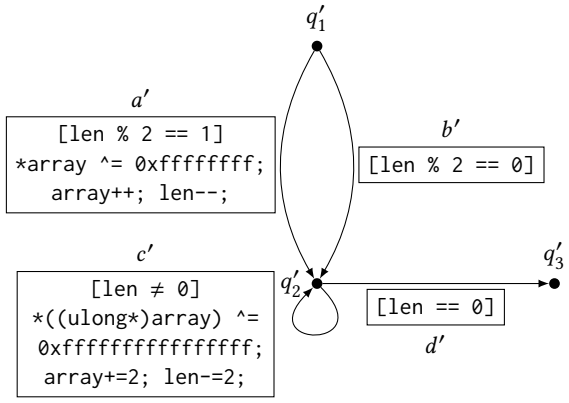
We begin by guessing an *alignment predicate*, ξ, over pairs of machine states from f and g that will help us align traces of f and g when run on the same input. For this example, consider the alignment predicate ξ = {`array + 4i = array'`}. Suppose ρ and ρ' are traces of f and g for a particular test case. We consider a machine state σ from ρ and σ' from ρ'. If the predicate $\xi(\sigma, \sigma')$ holds, we say the two traces are

```
1 void f(int* array, uint len) {
2   for(uint i = 0; i < len; i++)
3     array[i] ^= 0xffffffff;
4 }
```



(a) C source and CFG for the unoptimized program, f.

```
1 void g(int* array, uint len) {
2   if(len % 2 == 1) {
3     *array ^= 0xffffffff;
4     array++;
5     len--;
6   }
7   while(len) {
8     *((long*)array) ^= 0xffffffffffffffff;
9     array += 2;
10    len -= 2;
11  }
12 }
```



(b) C source and CFG for the vectorized program, g.

Figure 2. Functions f and g used in the example.

aligned by ξ at that pair of states. Additionally we consider ρ, ρ' to be aligned at the beginning and at the end, even if ξ does not hold. In Figure 4 we have drawn edges between every pair of states in the two traces that are aligned by ξ. Observe that the alignment may pair states in a many-to-many correspondence, and that edges may cross. A *trace alignment* by ξ is obtained by performing this procedure for a set of test cases.

The trace alignment gives pairs of *corresponding paths* that relate the behavior of f with g as follows. Consider any two

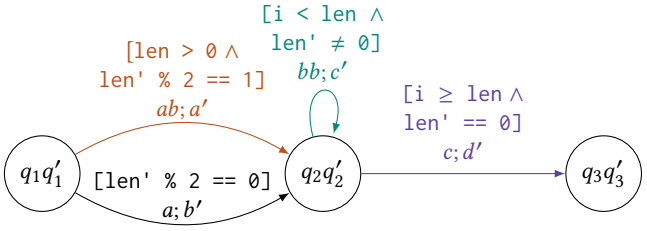


Figure 3. Simplified program alignment automaton for the example. The colors show the correspondence between the transitions and the pairs of corresponding paths in Figure 5.

edges e_i, e_j in Figure 4 that do not cross each other and have no edges in between them (e.g. e_{10} is "between" e_{00} and e_{21}, while e_{64} and e_{73} cross each other). Each of e_i, e_j is associated with a machine state in the execution trace of f. These two machine states delimit some series of basic blocks, called a *path*, in f. Similarly, e_i and e_j delimit a *corresponding path* in g. For example, consider edges e_{21} and e_{42}. Between index 2 and index 4 of the trace of f, the path bb is executed, while between index 1' and 2' of the trace of g, block c' is executed. Thus, bb and c' are corresponding paths. Figure 5 lists all such pairs of edges and the corresponding paths.

We use the corresponding paths to build a *program alignment automaton* (PAA) that (we hope) overapproximates the behaviors of both programs. The PAA has one node for each pair of program points. For each pair of corresponding paths in each pair of traces, we add a transition to the PAA labeled by these two paths. We perform a greedy simplification procedure to remove redundant nodes and edges (see Section 4.2). For the example, we remove the nodes q_2q_1', q_2q_3' and q_3q_2', and concatenate their incoming and outgoing transitions (so transitions $q_1q_1' \to q_2q_1'$ and transitions $q_2q_1' \to q_2q_2'$ are replaced by transitions $q_1q_1' \to q_2q_2'$). Note that different alignment predicates will define very different PAAs.

Figure 3 shows the simplified PAA for the example, which characterizes all the program behaviors. The three nodes are the natural outcome of the construction after simplification. This is in contrast to prior work such as [6, 7, 32] where corresponding points in the two programs, sometimes called *cutpoints*, must be chosen based on less information; usually cutpoints are chosen syntactically. Our construction guarantees that ξ holds in all nodes of the PAA (except possibly the entry and exit nodes) for all the traces generated by the test cases.

In Figure 3 there are two transitions between q_1q_1' and q_2q_2'; one is for inputs where `len` is odd, and a' is executed in g. The other is for inputs where `len` is even, and b' is executed in g instead (this transition comes from corresponding paths of aligned traces where the starting value for `len` is even). The paths labeling the transitions show that b is executed in f an extra time if a' is executed. Each path P has a path condition ψ_P, which is the conjunction of the predicates on

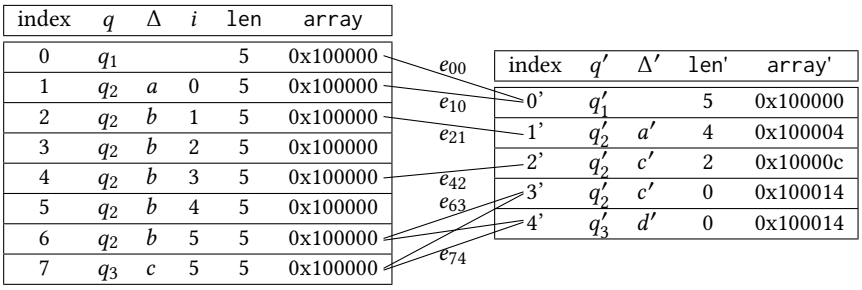


Figure 4. Execution traces of f and g for a particular input. Column q shows the program point and column Δ shows the last basic block executed. The edges indicate pairs of states where the alignment predicate `array + 4i = array'` holds.

Edges	States	P	Q
$e_{00} \to e_{10}$	$q_1q'_1 \to q_2q'_1$	a	ϵ
$e_{10} \to e_{21}$	$q_2q'_1 \to q_2q'_2$	b	a'
$e_{21} \to e_{42}$	$q_2q'_2 \to q_2q'_2$	bb	c'
$e_{42} \to e_{63}$	$q_2q'_2 \to q_2q'_2$	bb	c'
$e_{63} \to e_{64}$	$q_2q'_2 \to q_2q'_3$	ϵ	d'
$e_{63} \to e_{73}$	$q_2q'_2 \to q_3q'_2$	c	ϵ
$e_{64} \to e_{74}$	$q_2q'_3 \to q_3q'_3$	c	ϵ
$e_{73} \to e_{74}$	$q_3q'_2 \to q_3q'_3$	ϵ	d'

Figure 5. Pairs of edges and corresponding paths.

$$
\begin{aligned}
\phi_{q_1q'_1} &:= array = array' \wedge len = len' \wedge \omega = \omega' \\
\phi_{q_2q'_2} &:= array' - 4i = array \wedge len - i = len' \wedge \\
&\quad\ i \leq len \wedge \omega = \omega' \\
\phi_{q_3q'_3} &:= \omega = \omega'
\end{aligned}
$$

Figure 6. Invariants needed for the example. ω and ω' denote heap states of f and g. The alignment also allows us to show that $len' \equiv 0 \pmod 2$ at $q_2q'_2$, although this fact is unneeded.

its basic blocks. Each transition λ labeled by paths P, Q has a path condition given by $\psi_\lambda = \psi_P \wedge \psi_Q$, as shown in Figure 3.

Our next goal is to learn an invariant ϕ_s at every node s in the PAA and then prove that the PAA soundly overapproximates both programs. At the start node we fix the invariant to assert equality of the input registers and the initial heaps. At the exit, we assert equality of the output registers and the final heaps. The other invariants are learned using the execution traces from the test cases provided by the user (Section 4.4). A subset of the learned invariants for the example is shown in Figure 6.

The choice of alignment predicate is crucial to finding invariants. For example, suppose that our alignment predicate depicted in Figure 4 also paired state 3 of f's trace with state 1' of g's trace. After simplification, there is a transition $q_1q'_1 \to q_2q'_2$ labeled by $abb; a'$. Consequently, *none* of the invariants for $q_2q'_2$ depicted in Figure 6 would hold. Instead, to prove equivalence one would need a set of disjunctive invariants to reason about two cases: either f is one iteration ahead of g, or it is not (depending on whether the transition labeled $abb; a'$ is taken). A similar problem arises if one labels a transition $a; a'$ instead of $ab; a'$, as is the case in works such as [3, 32] where loop iterations are assumed to be in one-to-one correspondence.

To prove the equivalence of the two programs there are two primary types of proof obligations we must check (see Section 3.1). First, we check that the invariants hold. For each transition $s \to t$ with paths P and Q, we verify the following: if a pair of machine states satisfies ϕ_s, then if paths P and Q are executed, the execution terminates without error in states satisfying ϕ_t.

Second, we must ensure that the PAA has the necessary transitions to overapproximate all program behaviors. Each node s corresponds to a pair of program points (q_i, q'_j). We want to ensure that every pair of feasible execution paths starting at q_i and q'_j is represented in the automaton. Consider the node $q_2q'_2$. From node q_2 there are two kinds of executions: (α), those for which $i \geq len$ and execution halts; and (β), those for which $i < len$ and execution continues. From q'_2 there are similarly two kinds of executions: (γ), those for which $len' = 0$ and execution halts; and (δ), those for which $len' \neq 0$ and execution continues. Thus there are four pairs of possible behaviors: $\alpha\gamma, \alpha\delta, \beta\gamma$ and $\beta\delta$. Of these, $\alpha\gamma$ and $\beta\delta$ are already represented in the PAA via the self-loop at $q_2q'_2$ and the transition $q_2q'_2 \to q_3q'_3$. For the other two, we need to show they are infeasible. Now, $\alpha\delta$ only executes if $i \geq len$ and $len' \neq 0$, however $i \geq len \wedge len' \neq 0 \wedge \phi_{q_2q'_2}$ is unsatisfiable. Similarly $i < len \wedge len' = 0 \wedge \phi_{q_2q'_2}$ is also unsatisfiable, so $\beta\gamma$ is infeasible.

Verifying these proof obligations is sufficient to conclude that these two programs are equivalent for all inputs.

3 Formalization

We say that two `x86-64` functions, f and g, are *equivalent* if, when run starting in identical machine states (registers, stack, heap), one of the following holds:

1. both terminate normally, with identical heap-state and identical output registers; or
2. each program either encounters a run-time error or loops forever.

We use σ to denote a machine state, including the program counter, and all register, stack and heap values. We use ω to denote just the heap. When we use x to denote a state element of f, we use x' to denote the corresponding state element of g. A trace, ρ, is a sequence of machine states.

We use relational Hoare triples [5, 19] to express proof obligations. Let ϕ_1, ϕ_2 be predicates on pairs of machine states from f and g, let P (resp. Q) be a path through f (resp. g). Then $\{\phi_1\}\ P\ ;\ Q\ \{\phi_2\}$ denotes the statement: if $\phi_1(\sigma, \sigma')$ holds for states σ, σ' of f, g and paths P, Q are executed (implying the path conditions hold), then execution terminates normally in states σ'', σ''' where $\phi_2(\sigma'', \sigma''')$ hold.

A PAA is an automaton where each node s is labeled with a pair of program points, one from f and one from g, along with an invariant ϕ_s. We assume that f and g each have unique entry and exit program points. The start node of the PAA corresponds to the pair of program entries, and the unique final node corresponds to the pair of exit points. Each transition is labeled by a finite path in each program: a transition $(u, u') \to (v, v')$ must be labeled with a path P in f from u to v, and a path Q in g from u' to v' where either P or Q must be nonempty. The PAA fully determines a product program, although we do not explicitly build the product program in our work.

3.1 Proof Obligations for Program Alignment Automata

To use a PAA to check program equivalence, the following properties must be verified:

1. For each transition $s \to t$ labeled with paths P, Q it holds that $\{\phi_s\}\ P\ ;\ Q\ \{\phi_t\}$.
2. For each node $s = (u, u')$ all pairs of program paths through f and g starting from u and u' not included in the PAA must be infeasible. That is, if P and Q are paths through f and g starting at u and u', and there is no transition $s \to t$ labeled by P^*, Q^* where P^* is a prefix of P and Q^* is a prefix of Q, then $\{\phi_s\}\ P\ ;\ Q\ \{\text{false}\}$.
3. The PAA has no cycles of transitions where all the paths through f or g are empty.
4. The invariant of the final node implies that the heap states and output registers are equal.

The alignment predicate, and the trace alignment derived therefrom, play the critical role of selecting the right transitions for the PAA so that we can prove the invariants at each node. The following lemma illustrates the key inductive argument for a proof of equivalence, and the corollary establishes soundness.

Lemma 3.1. *Let A be a program alignment automaton where the above proof obligations have been checked. Suppose f and g are executed from states σ, σ' at the program points (u, u'), and there is a node $s = (u, u')$ of A where $\phi_s(\sigma, \sigma')$ holds. Then if f executes to completion without exceptions within m steps (each step is an execution of a basic block), g also executes to completion without exceptions, and their final states satisfy the invariant of the final node of A.*

Proof. By strong induction on m. When $m = 0$, the premises imply that f and g have already executed to completion and the conclusion holds. Suppose the lemma holds for $0 \leq i < m$. Assume f and g are executed from (u, u') and that f terminates within m steps. By proof obligation 2, some prefix of the execution traces of f and g must match the paths P, Q of some transition $\lambda : s \to t$ in A. Removing these prefixes from the execution traces, we now have a new pair of traces where f and g execute from t with states σ'', σ''' that satisfy ϕ_t (by proof obligation 1). In the case where P is non-empty f still executes to completion, but now within $j < m$ steps. By the inductive hypothesis, we can conclude the lemma holds. In the case where $P = \epsilon$, we repeat the step of identifying a matching transition and removing the trace k times, where k is the length of the longest series of transitions from s where the paths for f are empty; by proof obligation 3, k must be defined. Then proceed as before. □

```
function VERIFY(f, g, data)
    (d_train, d_test) ← Partition(data)
    AP ← GuessAlignmentPredicates(f, g, d_train)
    for all ξ ∈ AP do
        TA ← BuildTraceAlignment(ξ, d_train)
        A ← BuildPAA(TA)
        if TestPAA(A, d_test) then
            A ← LearnInvariants(A, d_test ∪ d_train)
            if CheckProofObligations(A) then
                return equivalent
            end if
        end if
    end for
    return unknown
end function
```

Figure 7. The equivalence checking algorithm.

Corollary 3.2 (Soundness)**.** *If there exists a program alignment automaton, A, for f, g where the proof obligations hold then f and g are equivalent.*

Proof. Suppose we run f, g on an input. By Lemma 3.1 if f terminates without error then g does also; by swapping f and g the converse also holds. The lemma also implies the final invariant of A holds, and by the fourth proof obligation f and g are equivalent. □

4 Equivalence Checking Procedure

Figure 7 gives pseudocode for our algorithm. The user supplies two functions, f and g, along with a set of test cases, *data*. The test cases are partitioned into two sets, a training set and a test set. We invoke `GuessAlignmentPredicates` with the training data to build a set of candidate alignment predicates (Section 4.6). For each alignment predicate ξ we call `BuildTraceAlignment` to construct a trace alignment, TA, over the training data (Section 4.1) and then use TA to construct the PAA (Section 4.2). To ensure that the PAA

is general and not overfitted to the training data, we use the test data to check the *viability* of the PAA via `TestPAA` (Section 4.3). Finally we learn the invariants for the PAA (Section 4.4) and check the proof obligations (Section 4.5).

4.1 Construction of the Trace Alignment

Given an alignment predicate ξ we construct a trace alignment. The trace alignment TA is a set of pairs (ρ, ρ') where ρ and ρ' are prefixes of the traces of f and g for some test case. We initialize TA to the empty set. For each training test case τ we execute f and g to obtain traces $\rho_\tau = \sigma_1\sigma_2\cdots\sigma_n$ and $\rho'_\tau = \sigma'_1\sigma'_2\cdots\sigma'_m$. For each σ_i, σ'_j we check $\xi(\sigma_i, \sigma'_j)$; when satisfied, we add the pair $(\sigma_1\sigma_2\cdots\sigma_i, \sigma'_1\sigma'_2\cdots\sigma'_j)$ to TA. Figure 4 shows prefixes of traces that are aligned by ξ in the example.

4.2 Construction of the Program Alignment Automaton

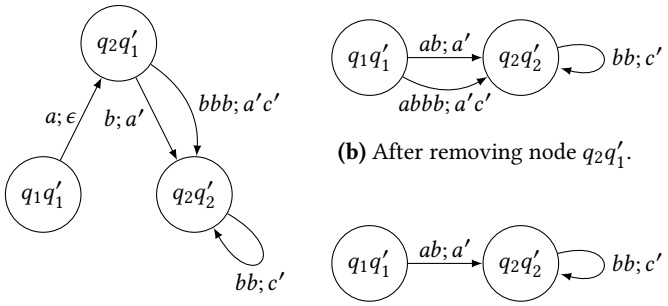
We initialize the PAA with a node for every pair of program points in the two programs. We consider pairs $(\rho, \rho') \in TA$ along with minimal v, v' such that $(\rho v, \rho' v') \in TA$ (e.g. for the trace alignment in Figure 4, we consider the pairs shown in Figure 5). For each such pair we add a transition $(p, p') \to (q, q')$ labeled by the paths of basic blocks taken by v and v', where (p, p') is the last pair of program points in ρ, ρ' and (q, q') is the last pair of program points in v, v'. As an optimization, we consider only v, v' that are small, for example, fewer than 10 machine states in length.

The PAA can be regarded as a nondeterministic finite automaton (NFA) in the following sense. A pair of traces ρ, ρ' for f and g is *accepted* by the PAA if there is a series of transitions (a *run* through the PAA) from the start node to the exit node that correspond with ρ, ρ' (i.e. concatenating the labels of the transitions gives paths that match paths taken by ρ and ρ'). The above construction ensures that every pair of traces in the training set is accepted by the PAA (we say the PAA "accepts the training set").

After performing this construction, we simplify the PAA by removing nodes and transitions while ensuring the PAA still accepts the training set. Removing nodes makes finding provably correct invariants easier, and removing transitions decreases the number of proof obligations. In our experiments, we find simplification reduces the number of nodes by 3.9x and the number of edges by 3.7x. We perform the following two operations until we reach a fixed point.

First, we remove every node s that does not have a self-loop other than the entry and exit. Suppose s has incoming transitions $r_1 \to s, \ldots, r_n \to s$ and outgoing transitions $s \to t_1, \ldots, s \to t_m$. For each i, j-pair replace the transitions $r_i \to s$ and $s \to t_j$ with a transition $r_i \to t_j$ labeled with the concatenation of the paths of the original two transitions.

Second, we remove extra transitions. If a transition $\lambda : s \to t$ is labeled with paths P, Q and transition $\lambda^* : s \to u$ is labeled with paths P^*, Q^* where P is a prefix of P^* and Q is a prefix of Q^*, then λ^* is removed. Once we cannot remove any more transitions or nodes, the PAA is simplified.



(a) PAA before simplification. **(b)** After removing node $q_2q'_1$. **(c)** After deleting the extra edge.

Figure 8. Example of simplification procedure.

Figure 8a shows a hypothetical PAA for the example (Section 2). We can remove node $q_2q'_1$ since it has no self loops. We combine the transition $a;\epsilon$ with each of $b;a'$ and $bbb;a'c'$ to get two transitions $q_1q'_1 \to q_2q'_2$ as shown in Figure 8b. Because ab is a prefix of $abbb$ and a' is a prefix of $a'c'$ we can remove one more transition to obtain the simplified PAA shown in Figure 8c.

4.3 Testing the Program Alignment Automaton

If the alignment predicate is chosen poorly, the PAA constructed in Section 4.2 may be overfitted to the training data. As a worst case example, consider the alignment predicate ξ = "false". Traces will only be aligned by ξ at the beginning and the end. Every new test case may add a new transition from the start node to the end node labeled by the entire pair of traces. Thus, there is no limit on the number of transitions (if the traces can be arbitrarily long) and such a PAA is not useful for equivalence checking. A good alignment predicate, on the other hand, results in a PAA to which no further transitions are needed to accept additional test cases — the PAA already captures all possible pairs of executions of the two programs. We must verify that the PAA is such a sound overapproximation of the two programs as part of equivalence checking (see Section 4.5), but we can eliminate many PAAs earlier by testing. We use a separate *test set* of inputs for this purpose.

By construction, the PAA accepts the training set (Section 4.2), so we check that the PAA also accepts the test set. This check is similar to the standard language membership test for NFAs. If the PAA fails to accept the test set, then we reject the PAA and try another alignment predicate.

4.4 Learning Invariants

Our goal is to learn invariants for each node of the PAA. We take a data-driven approach and use the test cases to guess a conjunction of predicates for each node, and later (Section 4.5) we discard the conjuncts that cannot be proven.

$$
\begin{aligned}
Inv \quad \rightarrow \quad & \sum c_i v_i = c \;\Big|\; v_1 - v_2 \le c \;\Big|\; \pm v \le c \\
& \Big|\; m = v \;\Big|\; v_1 - v_2 \equiv c_1 \pmod{c_2} \\
& \Big|\; v \equiv c_1 \pmod{c_2} \;\Big|\; \omega_{\overline{S}} = \omega'_{\overline{S}}
\end{aligned}
$$

Figure 9. The language of invariants. Each c represents a bitvector constant and m represents a memory location. v represents a register, a subregister, or a stack-allocated memory location. $\omega_{\overline{S}}$ represents the heap excluding the set of memory locations S.

First, for each node s of the PAA we need to identify a set of pairs of machine states, Σ_s, over which to learn invariants. For the traces of each test case (from either the test set or the training set), we consider every possible run of the PAA and record the machine states at each transition. Given a test case τ we run both programs to obtain traces ρ, ρ'. Let ν, ν' be prefixes of ρ, ρ' that execute paths P, Q. Consider every sequence of transitions (if any) from the start node of the PAA such that the concatenation of the labels of the transitions match P and Q. For each such sequence of transitions that ends in node s, we add the pair (σ, σ') to Σ_s where σ and σ' are the last machine states of ν and ν'.

The language of invariants, shown in Figure 9, includes linear equalities, inequalities and equalities mod n. Inequalities are needed for reasoning about branch conditions, and equalities mod n are needed to prove properties of warm-up and cool-down loops in vectorized code. There are three different data-driven techniques for learning the conjuncts.

First, for inequalities, we sample a subset of the data and find all the inequalities with the strongest bound possible. Then, we check if these inequalities hold over the entire data set; the failing ones are discarded.

Second, we use techniques from linear algebra to find a space of all linear equalities that hold over the data [32]. We construct a matrix M over the ring of 64-bit bitvectors $\mathbb{Z}_{2^{64}}$ containing program values where row i corresponds to σ_i and σ'_i, and column j corresponds to a register or stack location. We use SageMath version 7.5.1 [36] to compute the kernel $K = \ker M$. Each vector in the generating set for K corresponds to a linear equality that holds over all pairs (σ_i, σ'_i). Performing this computation over $\mathbb{Z}_{2^{64}}$ rather than $\mathbb{Z}$ is expensive, but necessary because some equalities hold over $\mathbb{Z}_{2^{64}}$ that do not hold over $\mathbb{Z}$ (in past work [6, 32] the invariant learning routine would miss some of these equalities). Therefore, we perform two optimizations. First, we do a pre-pass in which we remove pairs of columns where a linear relationship of the form $c_1 v_1 + c_2 v_2 = 0$ can be readily found. Second, we only sample $k = 25$ rows of test data for the matrix. We test the learned invariants against the rest of the data set; if the learned invariants do not hold, we sample up to k more rows from among the failures and repeat.

Lastly, for the remaining classes of invariants we learn the strongest possible invariant over the entire data set. Here, no division between test and training data is needed. We attempt to learn an equality mod n for every pair of program values. For each pair v_1, v_2 we compute all differences $d_i = v_1 - v'_2$ for each σ_i, σ'_i. We then compute the greatest common divisor d of all $d_i - d_j$. If $d \neq 1$, then we can find c such that $v_1 - v_2 \equiv c \pmod{d}$. To learn an invariant of the form $\omega_{\overline{S}} = \omega'_{\overline{S}}$ we choose the minimal set S for which the invariant holds on all test cases.

Invariants learned for the PAA in Figure 3 are shown in Figure 6.

4.5 Verifying Proof Obligations

We perform a Houdini-style [15] fixed point computation to reduce the set of learned invariants to those that can be proven by induction. For each node s we have the invariant $\phi_s = \phi_s^1 \wedge \cdots \wedge \phi_s^n$. For each transition $\lambda : t \rightarrow s$ labeled by paths P and Q we attempt to prove $\{\phi_t\}\ P\ ;\ Q\ \{\phi_s^i\}$ for $1 \le i \le n$. If any conjunct does not hold, we remove it from the invariant. We repeat this procedure until all the proofs succeed. We then check the remaining proof obligations (Section 3.1), and Corollary 3.2 implies equivalence.

Our implementation supports two ways to model the stack. The first models the stack conservatively, where we assume that the stack pointer is an arbitrary address that could alias with arbitrary data structures on the heap, and ensures that the two functions behave identically. However, for verifying optimizations that transform the stack, we also support assuming that stack locations do not alias with any heap locations or pointers in input parameters. In these cases we also assume that stack accesses of different sizes do not alias, so we model them using separate memory stores [39].

4.6 Space of Alignment Predicates

In practice we find there is a small space of predicates that almost always contains a useful alignment predicate for pairs of equivalent `x86-64` functions. Namely, choosing a predicate of the form $(c_1 v_1 - c_2 v_2 = k) \wedge \omega = \omega'$ is typically sufficient. Here v_1 and v_2 are registers or stack-allocated locations in f and g. We restrict $c_1, c_2 \in \{1, 2, 4, 8, 16\}$ and $k \in \mathbb{Z}$. Moreover, we only need to consider alignment predicates where either $c_1 = 1$ or $c_2 = 1$. There are 16 registers, but we only need consider registers whose values are defined (as determined by a program analysis). Thus, the total number of choices for c_1, c_2, v_1 and v_2 has a relatively small bound. For each of these, we heuristically pick k by finding values seen for $c_1 v_1 - c_2 v_2$ across different states that are in common across multiple pairs of traces. Performing this search is generally quite fast, and we make no attempt to rank the alignment predicates heuristically or try them in a particular order. If all the alignment predicates fail, we additionally attempt to use predicates of the form $c_1 v_1 - c_2 v_2 = k$, where

the alignment predicate does not constrain the heap states. We offer intuition for, and evaluate the utility of, this space of alignment predicates in Section 5.5.

5 Evaluation

In this section we seek to validate the following points:

- Our technique is able to verify the correctness of vectorization and other complex loop transformations as performed by modern compilers on `x86-64`. (Sections 5.1 and 5.2)
- Our technique can verify optimizations that are beyond the scope of existing automated black-box techniques. (Section 5.3)
- Our technique can verify equivalence checking benchmarks used to evaluate other state-of-the-art tools. (Section 5.4)
- The search space of alignment predicates that we use is suitable for realistic verification problems. (Section 5.5)

We conclude with limitations in Section 5.6.

5.1 Experimental Setup

To evaluate our method we construct a set of benchmarks for verifying vectorization optimizations. We started with 156 functions from the *Test Suite for Vectorizing Compilers* (TSVC), which was developed "to assess the vectorizing capabilities of compilers" and ported to C in [24]. We removed five classes of functions from the original TSVC set:

- Functions that could not be vectorized using `-msse4.2` and `-O3` with either `gcc` 4.9.2 or `clang` 3.4. These functions are not interesting in our evaluation because the loop structures are preserved. These functions should be easy for both our technique and other state-of-the-art tools. (96 functions)
- Duplicate functions. Some TSVC functions were designed to check that a compiler could perform an analysis to verify the safety of an optimization; however, after successful vectorization, the generated `x86-64` code matches that of another function. (6 functions)
- Functions with method calls. Our implementation does not support method invocations. (9 functions)
- Out of scope functions designed to test loop interchange. See Section 5.6. (6 functions)
- Functions with two-dimensional arrays or memory indirection. (11 functions)

The TSVC functions operate on statically-allocated, fixed-size global arrays of floating point values. While our technique works as is on over 80% of the floating point benchmarks (by using uninterpreted functions to model floating point operations), there are additional issues when learning invariants from floating point data that are not addressed by existing invariant inference techniques. For example, there are multiple binary representations for some floating point values, such as `NaN`. These issues are orthogonal to our contributions; to separate the evaluation of our method from the details of floating point semantics, we systematically replaced floating point types with integer types. We constructed 256 test cases by creating machine states containing input arrays of randomly-chosen bytes. This same set of test cases was sufficient to obtain code coverage over all these benchmarks. We also added a parameter to each function to specify the array length. We added assumptions on the input values to prevent pointers for different arrays from aliasing. Adding assumptions to avoid undefined behavior is generally required for equivalence checking [8, 33].

We were left with 28 functions. Most iterate over one or more arrays (up to 5), perform arithmetic, and update the arrays. Some process the array forwards, some backwards, and some with a stride. Some have loop carried dependencies, others do not. One function, `s176`, features a doubly-nested loop. No combination of these features hindered our ability to check equivalence. For each of the 28 functions we attempted to prove that `gcc -O1` code was equivalent to `gcc -O3` code and to `clang -O3` code, resulting in a total of 56 benchmarks.

Sometimes discharging a particular proof obligation takes a long time or times out using one SMT solver, but finishes quickly with another solver. Thus we use two solvers, `Z3` [11] (commit `7f6ef0b6`) and `CVC4-1.5` [2] with the theory of arrays and bitvectors. Also, the encoding of constraints that represent memory accesses may have a profound impact on solver performance. Therefore we implement two memory models [39], a flat memory model, and one based on alias relationship mining (ARM). The flat model encodes all memory accesses as a read or an update to an array (with separate arrays for the stack, if needed). ARM uses data from test cases to guess and prove relationships that ensure pointers do not alias; then the constraints are encoded with minimal use of arrays [6]. We set a 30-minute timeout for each proof obligation for each solver and memory model. We use the result of whichever solver and memory model pair finishes first. We use the counterexamples from the SMT solver to eliminate other proof obligations that are demonstrably false, as in [16].

We used rigorously tested semantic models for `x86-64` instructions developed by hand [6] and synthesized automatically [18]. We model multiplications and floating point operations using uninterpreted functions. We performed the construction of the PAA for each benchmark using one core of an Intel Xeon E5-2667 CPU @ 3.3GHz machine. We use a pool of preemptible cloud virtual machines to check the proof obligations.

Our implementation is available at https://github.com/bchurchill/pldi19-equivalence-checker and the latest citable release may always be found at https://doi.org/10.5281/zenodo.2646657.

Benchmark	gcc -O1 LOC	gcc -O3 LOC	gcc -O3 Out	clang -O3 LOC	clang -O3 Out
s000	14	18	✓	44	✓
s1112	12	31	✓	59	✓
s112	14	55	✓	24	✓NV
s121	16	44	✓	48	✓
s1221	14	24	✓	37	✓
s122	17	108	✓S	21	✓NV
s1251	18	29	✓	60	✓
s127	22	82	✓	31	✓
s1281	21	30	✓	66	✓
s1351	12	17	✓	51	✓
s162	17	49	✓	58	✓
s173	17	56	✓	70	✓S
s176	29	99	✓S	34	✓NV
s2244	19	56	✓	65	✓
s243	25	30	✓NV	68	✓
s251	16	27	✓	49	✓
s3251	22	149	✓S	26	✓NV
s351	29	130	×S	24	✓
s452	22	27	✓	25	✓
s453	15	22	✓	15	✓NV
sum1d	15	28	✓	45	✓
vdotr	17	28	✓	49	✓
vpvpv	15	26	✓	38	✓
vpv	13	25	✓	37	✓
vpvts	14	30	✓S	54	✓
vpvtv	14	26	✓	36	✓
vtv	14	25	✓	36	✓
vtvtv	15	26	✓	51	✓

Figure 10. Results for 56 vectorization benchmarks. ✓ represents successful verification and × represents a timeout. For six functions only one compiler succeeds in vectorization; these benchmarks are marked by NV. Benchmarks requiring assumptions about the stack are marked by S.

5.2 Results

A list of the benchmarks and the outcomes are shown in Figure 10. We successfully verified 55 of the 56 benchmarks. The one failure (`s351-gcc`) was due to a timeout. In all other cases the proofs succeeded. The PAAs all had 3 or 4 nodes. The number of edges varied from 4 to 254, with a median of 4 and an average of 9. The number of conjuncts in the invariants in the final PAA ranged from 374 to 1417, with a median of 651. The median time to discharge all proof obligations was 45.0 CPU hours; the minimum time was 2.5 CPU hours (`s112-clang`) and the maximum 1166 CPU hours (`s351-clang`). The end-to-end time for this benchmark using the cloud was 4.6 hours. The cost for cloud instances was $0.01 per CPU hour, so the cost of checking the proof obligations for this benchmark was $11.66 while a typical problem cost just $0.45.

The most difficult benchmark, `s351`, includes a loop with five multiplications, five additions and ten memory dereferences in each iteration. The `s351-gcc` benchmark, which encountered timeouts while checking proof obligations, included a 4-way vectorized loop with a fully-unrolled cooldown loop to handle the last four iterations. It is likely that with more effort our constraint generation procedure can be tuned to discharge the problematic proof obligations more efficiently. While `s351-clang` still used vector instructions, `clang` generated much simpler code than `gcc`. Still, the `s351-clang` benchmark took the most CPU time of all the successful benchmarks.

5.3 GNU C Library `strlen` Case Study

Sometimes compilers are unable to vectorize performance-critical functions and so library developers perform the vectorization themselves. This is the case for the `strlen` function in `glibc`, which was most recently updated in May 2009 with the release of version 2.10.1. There is a test in the `glibc` test suite that runs both a reference implementation and the hand optimized one, and checks that the outputs are equal. Instead of running the programs on some inputs, we can leverage test cases to prove that the two implementations are equivalent for all inputs. We successfully verified the correctness of the `strlen` function (shown in Figure 11a) originally released in 2.10.1, which still ships as of 2019 in version 2.29, against a simple reference implementation (Figure 11b). The alignment predicate found asserts the equality of the pointers into the string (`ptr` and `p`). The end-to-end verification time was only 3.3 minutes on a single CPU core.

The vectorized code has two loops; the warm-up loop (lines 6-8) counts characters one-by-one until the pointer reaches an 8-byte boundary or a null character. The main loop (lines 14-29) reads 8 characters from the string at a time and uses clever bit-manipulation techniques to check if any of the 8 characters are null. If so, the code checks the remaining characters one-by-one and returns the length; otherwise, the loop continues.

Thus, the code reads beyond the end of the string unless the string ends at an 8-byte boundary or the warm-up loop encounters the null terminator. This is safe on `x86-64` because memory permissions are set on a page-level granularity (usually 4kB in size). If m is a memory address the process is allowed to read, so is $8\lfloor m/8 \rfloor + 7$. While the optimized code can perform an out of bounds read, it never uses this value, and the read does not trigger a page fault (assuming that the unoptimized code does not fault). This example shows two programs that are provably equivalent, even though they dereference a different set of memory locations.

In general, if the memory locations accessed by f are provably on the same pages as those accessed by g, then f raises a page fault if and only if g does; but, if the memory

```
1 size_t strlen (char *str) {
2   char *ptr;
3   ulong *longword_ptr;
4   ulong longword, himagic, lomagic;
5
6   for (ptr = str; ((ulong) ptr & 7) != 0; ++ptr)
7     if (*ptr == '\0')
8       return ptr - str;
9
10  longword_ptr = (ulong *) ptr;
11  himagic = 0x8080808080808080L;
12  lomagic = 0x0101010101010101L;
13
14  for (;;)
15  {
16    longword = *longword_ptr++;
17    if ((longword - lomagic) & ~longword & himagic)
18    {
19      char *cp = (char*)(longword_ptr - 1);
20      if (cp[0] == 0) return cp - str;
21      if (cp[1] == 0) return cp - str + 1;
22      if (cp[2] == 0) return cp - str + 2;
23      if (cp[3] == 0) return cp - str + 3;
24      if (cp[4] == 0) return cp - str + 4;
25      if (cp[5] == 0) return cp - str + 5;
26      if (cp[6] == 0) return cp - str + 6;
27      if (cp[7] == 0) return cp - str + 7;
28    }
29  }
30 }
```

(a) Vectorized `strlen` implementation (simplified). The main loop has eight different branches to exit, and the warm-up loop has two. Compilation adds an extra branch that skips the warm-up loop. The alignment predicate ensures that each of these paths is mapped to the correct number of iterations in the reference implementation.

```
1 size_t strlen (char *s) {
2   char* p;
3   for(p = s; *p; ++p);
4   return p - s;
5 }
```

(b) Reference `strlen` implementation.

Figure 11. Two implementations of `strlen`.

accesses are on different pages, no such guarantee exists. For the sake of checking aggressive optimizations, we decided not to model page faults (we do, however, check for final heap equality, which addresses most faults due to memory writes). Thus $\{\phi_1\}\ P\ ;\ Q\ \{\phi_2\}$ may hold even if path P contains a memory access but path Q does not. There is no guarantee that equivalent programs will access memory pages in the same order; f could read and write a memory location in each loop iteration, while g reads and writes the memory location once (Section 5.4 offers one such example). Therefore, fully modeling page faults likely requires invariants that track which memory locations each program has accessed.

We also discovered that the hand-optimized code was written conservatively. When the guard of the if-statement on line 17 is satisfied, one of the eight return statements is always taken. We can optimize the code by moving the cascade of if-statements to the outside of the loop, and we proved this is sound.

To the best of our knowledge, no other black-box technique in the equivalence checking, relational verification nor translation validation literature is able to automatically verify this example. There are two challenging aspects to highlight. First, the number of iterations executed in the warm-up loop and main loops are data-dependent; i.e., the number of iterations of the warm-up loop depends on the alignment of the input string to an 8-byte boundary. Second, the PAA has a large number of edges, and a naive search to build the PAA is too inefficient. Using an alignment predicate makes the search tractable.

5.4 Comparison with Related Work

We believe techniques that depend on syntactic alignment of the two programs [13, 14, 26, 27, 32] fail on most or all of our benchmarks, including at least 47 benchmarks where loop unrolling has been performed (usually as part of vectorization). In [3] the authors suggest unrolling one loop and then attempting a syntactic alignment. This approach does not support cool-down loops (present in 21 of our benchmarks) nor loop peeling optimizations (present in another 9 benchmarks). The technique of [7] succeeds on benchmarks unrolled μ times, where μ is an unroll factor. The cost of the technique is superexponential in μ in the worst case and reported results are only for $\mu = 1$ [7, 16]. Among our benchmarks, 32 have been unrolled 4 times and 15 have been unrolled 8 times. Finally we believe ours is the only black box, automated technique able to check equivalence for `glibc strlen` (Section 5.3) and our running example (Section 2).

A challenging equivalence checking problem is presented in [7]. As far as we know, only our technique and the technique of [7] are able to handle this problem. The benchmark consists of checking the correctness of a loop that sums the positive integers of an array after optimizations have been performed, including loop inversion, a transformation of branch conditions, replacing a branch inside the loop with a conditional move instruction, and register allocation. The unoptimized program writes to a global heap variable on every iteration, while the optimized version only writes the result once at the end of the loop. Their benchmark was for 32-bit `x86` rather than `x86-64`, but we found that compiling the C source on `x86-64` with `gcc 4.9.2` using `-O0` and `-O1` produced the same control flow graphs and the same optimizations; we believe that this modified benchmark is a suitable proxy for the original.

We successfully verified this benchmark; the alignment predicate we found related the stack-allocated pointer of the unoptimized program with an index counter in the optimized one and did not relate heap states. The total time to guess the alignment predicate, construct the PAA, learn invariants and verify the proof obligations on a single CPU core was 34.4 minutes. Most of the time was spent verifying the proof

obligations, which was done only using Z3 and only with the flat memory model.

The authors of [7] also demonstrate a large scale evaluation of their technique on whole binaries, but in whole programs many of the equivalence checks between corresponding functions are easy (e.g. do not involve loop optimizations), and [7] does not describe the harder equivalence checking problems. Since our contribution is about equivalence checking of loops, our evaluation focuses on loops rather than whole programs.

5.5 Search over Alignment Predicates

We performed an experiment to count the number of alignment predicates in the search space for each benchmark, and the number of viable PAAs that we could build (meaning the number of PAAs that accept the test set; see Section 4.3). For each benchmark we tried between 182 and 3318 alignment predicates, with a median of 1130. Between 0.37% and 22% of these alignment predicates led to viable PAAs. Averaging across the benchmarks, 3.1% of alignment predicates succeeded. At least 8 viable PAAs were found per benchmark, with a maximum of 65 and a median of 28. These findings suggest that our space of alignment predicates is robust for our set of benchmarks.

In practice the successful alignment predicates typically relate a pointer or counter in f with a pointer or counter in g. This is the case in our example (Section 2) where the alignment predicate matches the value of a pointer in f, namely `array + 4i`, with the pointer `array'` in g. When g processes 8 bytes of the array and the pointer increases by 8, we find a corresponding path in f where 8 bytes are processed and its pointer increases by 8. The powers of two in our alignment predicates arise because counters are generally multiplied by powers of two to address array locations, and not due to specifics of any optimizations performed on our benchmarks (e.g. the number of loop iterations unrolled). Alternatively, for the example, we can use equality of heap states as the alignment predicate to ensure that the memory writes of f and g are aligned and obtain the same result. For benchmarks where heap equality alone was a suitable alignment predicate, a large proportion of alignment predicates worked. We also observe that some alignment predicates succeed in aligning one loop an iteration (or k iterations) ahead of the other. We can check the proof obligations for the resulting PAAs as long as the invariants are able to sufficiently relate the program states despite this offset.

Since there are only a few ways to reference a memory location on `x86-64`, it is unsurprising that even our simple alignment predicates suffice to identify corresponding uses of pointers and counters between the two programs. For example, if f accesses an array using a pointer in register r_1, and g accesses an array of k-byte elements using a base address b and counter register r_2, then the alignment predicate $r_1 = b + k * r_2$ would assert the equality of these two memory dereferences. Indeed, this is in our space of alignment predicates.

While it did not arise in our benchmarks, we expect that some equivalence checking problems will require the alignment predicate to assert an equality over three or four registers. Four registers would be the maximum required to relate any two pointer dereferences. We also expect that some benchmarks involving multiple loops will require a disjunction over program points; for example, we may want to use one alignment predicate for one loop, and another alignment predicate for another loop. While one could extend our work to such problems by broadening the space of alignment predicates and thus increasing search times, our observation that good alignment predicates tend to relate pointers and counters suggests that these alignment predicates may be guessed directly from the program text. We leave this question to future work.

5.6 Limitations

A main limitation of our work is that we cannot reason about transformations that reorder an unbounded number of memory writes, for example, loop splitting, loop fusion, loop interchange and loop tiling optimizations. This is because the only invariants we learn and prove over the heap states assert heap equality on all but a finite set of memory locations. This limitation could be addressed by learning and proving more general quantified invariants over heap states.

Another limitation arises when the correspondence between the control flow of the two programs depends on an unbounded input. Consider the two functions in Figure 12 where the loop of f has been flattened. Here, m iterations in f correspond to 1 iteration in g. As far as we know, no equivalence checking techniques that construct a product program or similar structures are able to verify this benchmark as is (although those that summarize loops, like [10], may succeed). The reason is that the product program needs to align the entire execution of the inner loop of f with m iterations of the loop of g. To extend our approach to benchmarks like these, we would need to summarize loops (in this case the inner loop of f) and check for termination.

However, we confirmed our method can verify a modified version of this benchmark where other approaches using product programs likely fail. If the input value m is constrained to a small finite set $m \in \{c_1, c_2, \ldots, c_k\}$ while n is left unbounded then we can construct a PAA for the two programs and prove equivalence. The PAA contains a node s with k transitions $\lambda_i : s \to s$ where λ_i relates c_i iterations of f to 1 iteration of g. In essence, the PAA we learn creates a disjunction of all the k cases and we check each one. We can reason disjunctively because the path condition for λ_i only holds when $m = c_i$.

```
1 int f(uint n, uint m) {
2   int k = 0;
3   for(uint i = 0; i < n; ++i) {
4     for(uint j = 0; j < m; ++j) {
5       k++;
6     }
7   }
8   return k;
9 }
10 int g(uint n, uint m) {
11   int k = 0;
12   for(uint i = 0; i < n; ++i) {
13     k += m;
14   }
15   return k;
16 }
```

Figure 12. A difficult problem for equivalence checking via product programs.

6 Related Work

The most similar work to ours is that of [7], where the authors perform black-box equivalence checking across a variety of compiler optimizations. They construct and check a representation of a product program called a joint transfer function graph (JTFG) without any need for test cases or execution data. The equivalence checking method requires that branch conditions of one program provably match branch conditions of the other, which does not hold for either `strlen` (Section 5.3) or the example of Section 2. Also, paths in one program must correspond to sets of paths in the other, rather than allowing a many-to-many relationship. This assumption excludes a number of realistic benchmarks, such as `strlen`. Where the technique succeeds, the equality of branch conditions defines an alignment predicate which we can use to attempt equivalence checking; however, the converse does not hold. We believe this technique could be generalized by replacing the branch condition equality check with checking an alignment predicate.

Translation validation [26, 31, 34] uses compiler instrumentation to help generate a simulation relation to prove the correctness of compiler optimizations. A black box technique such as ours requires no effort to instrument the compiler or understand the specific transformations it performs, and further enables the comparison of programs produced by different people or compilers (as illustrated in our case study of `glibc strlen`).

In the literature of translation validation, equivalence checking, and relational verification, many works align loop iterations of one program in one-to-one correspondence with loop iterations of the other (as in Figure 1d), meaning that both programs execute each loop body the same number of times. Consequently, they often fail to prove equivalence of two functions where the loops execute for different numbers of iterations, which is the case for optimizations such as loop peeling, loop unrolling, and vectorization. Examples include past work on data-driven equivalence checking [32], Necula's well-known translation validation work [26], and others [13, 14, 27]. A related technique is to apply transformations (e.g. unrolling) to the programs so that the loop executions in the transformed programs are in correspondence, but this is brittle in the presence of unforeseen optimizations [3, 4]. In [23] the authors use a sequential composition of programs (as in Figure 1c), and attempt to summarize them by solving integer recurrences.

In contrast, the authors of [9, 10, 25] use constrained Horn clauses to summarize the entire execution of two programs and then use Horn clause solvers to prove equivalence. To make the problem tractable, the authors introduce a transformation called predicate pairing [9], where predicates from the two programs are combined into one predicate that models both programs. These predicates usually correspond to matching one iteration of a loop or one recursive step in one program with one in the other. The authors do not report on their ability to handle optimizations such as vectorization or loop unrolling where loop iterations or recursive steps are executed in one program more often than the other.

Some works use manually-provided specifications to perform alignment. For example, in [21] the authors prove the correctness of some difficult compiler optimizations, such as loop interchange, by using programmer-supplied templates which imply a correspondence between loop-free code fragments. Other works allow users to specify "control flow synchronization points" manually [20].

Equivalence checking techniques for affine programs can typically handle transformations that reorder loop iterations, such as loop tiling, loop interchange, vectorization and loop fusion, among others. In [38] the authors use abstract interpretation to summarize loops with a polyhedral domain. Polycheck [1] performs verification for affine programs dynamically; it uses a modified compiler to generate an instrumented binary that performs checks at run-time. The authors of [30] handle affine programs where loops have constant bounds; the authors expand traces of both programs and attempt to match operations between the traces via a series of normalization and rewriting steps. Similarly, the work [12] checks equivalence of loop parallelization and vectorization optimizations in a more general non-affine setting, but is limited to cases where the control flow of the program remains the same for all possible inputs.

In equality saturation [35, 37] authors build graphs of expressions for each program, transform the graphs via a series of rewrite rules, and check for equality. These approaches depend on the manually-specified rewrite rules and do not attempt to build a product program.

Some works use unsound reasoning for loops. SymDiff [17] models loops unsoundly by unrolling them for a

fixed number of iterations. UC-KLEE [28] explores a finite number of paths through a pair of programs to find differences. Recent work considers the correctness of peephole optimizations without loops [22], where the authors make progress on problems such as modeling undefined behavior.

7 Conclusion

In this paper we describe a semantic, data-driven approach to constructing product programs for equivalence checking. From an alignment predicate we construct a trace alignment and product program that enables verification. We demonstrate the applicability of this approach to a number of realistic benchmarks beyond the reach of prior equivalence checking work.

Acknowledgments

This work was supported by NSF grant CCF-1160904.

References

[1] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. PolyCheck: Dynamic Verification of Iteration Space Transformations on Affine Programs. In *Principles of Programming Languages (POPL '16)*. 539–554.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification (CAV '11)*. 171–177.

[3] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. 123–134.

[4] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *International Conference on Formal Methods (FM '11)*. 200–214.

[5] Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Principles of Programming Languages (POPL '04)*. 14–25.

[6] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 313–326.

[7] Manjeet Dahiya and Sorav Bansal. 2017. Black-Box Equivalence Checking Across Compiler Optimizations. In *Asian Symposium on Programing Languages and Systems (APLAS '17)*. 127–147.

[8] Manjeet Dahiya and Sorav Bansal. 2017. Modeling Undefined Behavior Semantics for Checking Equivalence Across Compiler Optimizations. In *Haifa Verification Conference (HVC '17)*.

[9] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2016. Relational Verification Through Horn Clause Transformation. In *International Static Analysis Symposium (SAS '16)*. 147–169.

[10] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2017. Enhancing Predicate Pairing with Abstraction for Relational Verification. In *Logic-Based Program Synthesis and Transformation (LOPSTR '17)*.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. 337–340.

[12] Sudakshina Dutta, Dipankar Sarkar, Arvind Rawat, and Kulwant Singh. 2016. Validation of Loop Parallelization and Loop Vectorization Transformations. In *Evaluation of Novel Software Approaches to Software Engineering (ENASE 2016)*. 195–202.

[13] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2015. Automated Discovery of Simulation Between Programs. In *Logic for Programming, Artificial Intelligence, and Reasoning*. 606–621.

[14] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Automated Software Engineering (ASE '14)*. 349–360.

[15] Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *International Symposium of Formal Methods Europe*. 500–517.

[16] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. 2018. Effective Use of SMT Solvers for Program Equivalence Checking Through Invariant-Sketching and Query-Decomposition. In *Theory and Applications of Satisfiability Testing (SAT '18)*.

[17] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow? Static Cross-version Compiler Validation. In *Foundations of Software Engineering (ESEC/FSE '13)*. 191–201.

[18] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Programming Language Design and Implementation (PLDI '16)*. 237–250.

[19] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[20] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. 2016. Relational Program Reasoning Using Compiler IR. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. 149–165.

[21] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Programming Language Design and Implementation (PLDI '09)*. 327–337.

[22] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Programming Language Design and Implementation (PLDI '15)*. 22–32.

[23] Nuno P. Lopes and José Monteiro. 2016. Automatic Equivalence Checking of Programs With Uninterpreted Functions and Integer Arithmetic. *International Journal on Software Tools for Technology Transfer* 18, 4 (Aug 2016), 359–374.

[24] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Parallel Architectures and Compilation Techniques (PACT '11)*. 372–382.

[25] Dmitry Mordvinov and Grigory Fedyukovich. 2017. Synchronizing Constrained Horn Clauses. In *LPAR*. 338–355.

[26] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Programming Language Design and Implementation (PLDI '00)*. 83–94.

[27] Nimrod Partush and Eran Yahav. 2013. Abstract Semantic Differencing for Numerical Programs. In *Static Analysis Symposium (SAS '13)*. 238–258.

[28] David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *Computer Aided Verification (CAV '11)*. 669–685.

[29] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. *SIGPLAN Not.* 48, 4 (2013), 305–316.

[30] Markus Schordan, Pei-Hung Lin, Dan Quinlan, and Louis-Noël Pouchet. 2014. Verification of Polyhedral Optimizations with Constant Loop Bounds in Finite State Space Computations. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*. 493–508.

[31] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Programming Language Design and Implementation (PLDI '13)*. 471–482.

[32] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-Driven Equivalence Checking. In *Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*. 391–406.

[33] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*. 147–162.

[34] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Computer Aided Verification (CAV '11)*. 737–742.

[35] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Principles of Programming Languages (POPL '09)*. 264–276.

[36] The Sage Developers. 2017. *SageMath, the Sage Mathematics Software System (Version 7.5.1)*. `http://www.sagemath.org`.

[37] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-Graph Translation Validation for LLVM. In *Programming Language Design and Implementation (PLDI '11)*. 295–305.

[38] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences. *ACM Transactions on Programing Languages and Systems (TOPLAS)* 34, 3 (2012), 1–35.

[39] Wei Wang, Clark Barrett, and Thomas Wies. 2017. Partitioned Memory Models for Program Analysis. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '17)*. 539–558.

[40] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods (FM '08)*. 35–51.