Data-Driven Equivalence Checking

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Eric Schkufza
Stanford University
eschkufz@cs.stanford.edu

Berkeley Churchill
Stanford University
bchurchill@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

We present a data driven algorithm for equivalence checking of two loops. The algorithm infers simulation relations using data from test runs. Once a candidate simulation relation has been obtained, off-the-shelf SMT solvers are used to check whether the simulation relation actually holds. The algorithm is sound: insufficient data will cause the proof to fail. We demonstrate a prototype implementation, called DDEC, of our algorithm, which is the first sound equivalence checker for loops written in x86 assembly.

Categories and Subject Descriptors D.1.2 [*Automatic Programming*]: Program Transformation; D.2.4 [*Program Verification*]: Correctness proofs; D.3.4 [*Processors*]: Compilers; D.3.4 [*Processors*]: Optimization

Keywords Binary Analysis; Compilers; Markov Chain Monte Carlo; Optimization; Superoptimization; SMT; Verification; x86

1. Introduction

Equivalence checking of loops is a fundamental problem with potentially significant applications, particularly in the area of compiler optimizations. Unfortunately, the current state of the art in equivalence checking is quite limited: given two assembly loops, no existing technique is capable of verifying equivalence automatically, even if they differ only in the application of standard loop optimizations. In this paper, we present the first practically useful, automatic, and sound equivalence checking engine for loops written in x86.

Proofs of equivalence at the binary level are more desirable than proofs at the source or RTL level, because such proofs minimize the trusted code base. For example, a bug in the code generator of a compiler can invalidate a proof performed at the RTL level or we can have a “what you see is not what you execute” (WYSINWYX) phenomenon [3] and the generated binary can deviate from what is intended in the source.

Existing techniques for proving equivalence can be classified into three categories: sound algorithms for loop-free code [1, 6, 10, 11, 23]; algorithms that analyze finite unwindings of loops or finite spaces of inputs [17, 20, 28, 30]; algorithms that require knowledge of the particular transformations used for turning one program into another [24, 36] and the order in which the transformations have been applied [14, 25, 29]. In contrast, our approach to handling equivalence checking of loops does not assume any knowledge about the optimizations performed.

We have implemented a prototype version of our algorithm in a system called DDEC (Data-Driven Equivalence Checker). This tool checks the equivalence of two loops written in x86 assembly. In outline the tool works as follows. First, DDEC guesses a *simulation relation* [25]. Roughly speaking, a simulation relation breaks two loops into a set of pairs of loop-free code fragments. Logical formulas associated with each pair describe the relationship of the input states of the fragments to the output states of the fragments. Second, DDEC generates verification conditions encoding the x86 instructions contained in each loop-free fragment as SMT [7] constraints. Finally, DDEC constructs queries which verify that the guessed relationships between the code fragments in fact hold. By construction, if the queries succeed they constitute an inductive proof of equivalence of the two loops.

It is worth stressing that DDEC works directly on unmodified binaries for x86 loops. The x86 instruction set is large, complex, and difficult to analyze statically. The key idea that makes DDEC effective in practice, and even sim-

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10…$15.00.
http://dx.doi.org/10.1145/2509136.2509509

ply feasible to build, is that the process of guessing a simulation relation (step 1 above) is constructed not via static code analyses, but by using data collected from test cases. Because DDEC is data driven, it is able to directly examine the precise net effect of code sequences without first going through a potentially lossy abstraction step. Of course, the use of test cases is an under-approximation and may not capture all possible loop behaviors. Nonetheless, DDEC is sound; a lack of test coverage may cause equivalence checking to fail, but it cannot result in the unsound conclusion that two loops are equivalent when they are not.

While our main result is simply a sound and effective approach to checking the equivalence of loops, we also show that sufficiently powerful equivalence checking tools such as DDEC have novel applications. In particular, we are able to verify the equivalence of binaries produced by completely different compiler toolchains. For a representative set of benchmarks, we automatically prove the equivalence of code produced by CompCert and `gcc` (with optimizations enabled), which allows us to certify the correctness of the entire `gcc` compilation or, conversely, to produce more performant CompCert code. Additionally, we extend the applicability of STOKE [33], a superoptimizer for straight-line programs, to loops. We replace STOKE's validator by DDEC and the resulting implementation is able to perform optimizations beyond STOKE's original capabilities, in fact producing verified code which is comparable in performance to `gcc -O3`.

DDEC is not without limitations. In particular, DDEC restricts the expressiveness of invariants required for a proof to be conjunctions of linear or nonlinear equalities. However, for most interesting intra-procedural optimizations, simple equalities appear to be sufficiently expressive [25, 32, 36]. DDEC is also currently unable to reason about floating point computations, simply because the current generation of off-the-shelf SMT solvers do not support floating point reasoning. Floating point reasoning is orthogonal to our contributions; when even limited floating point reasoning becomes available, DDEC can incorporate it immediately. However, the current state of the art limits the demonstration of DDEC to loops that work only with non-floating point values.

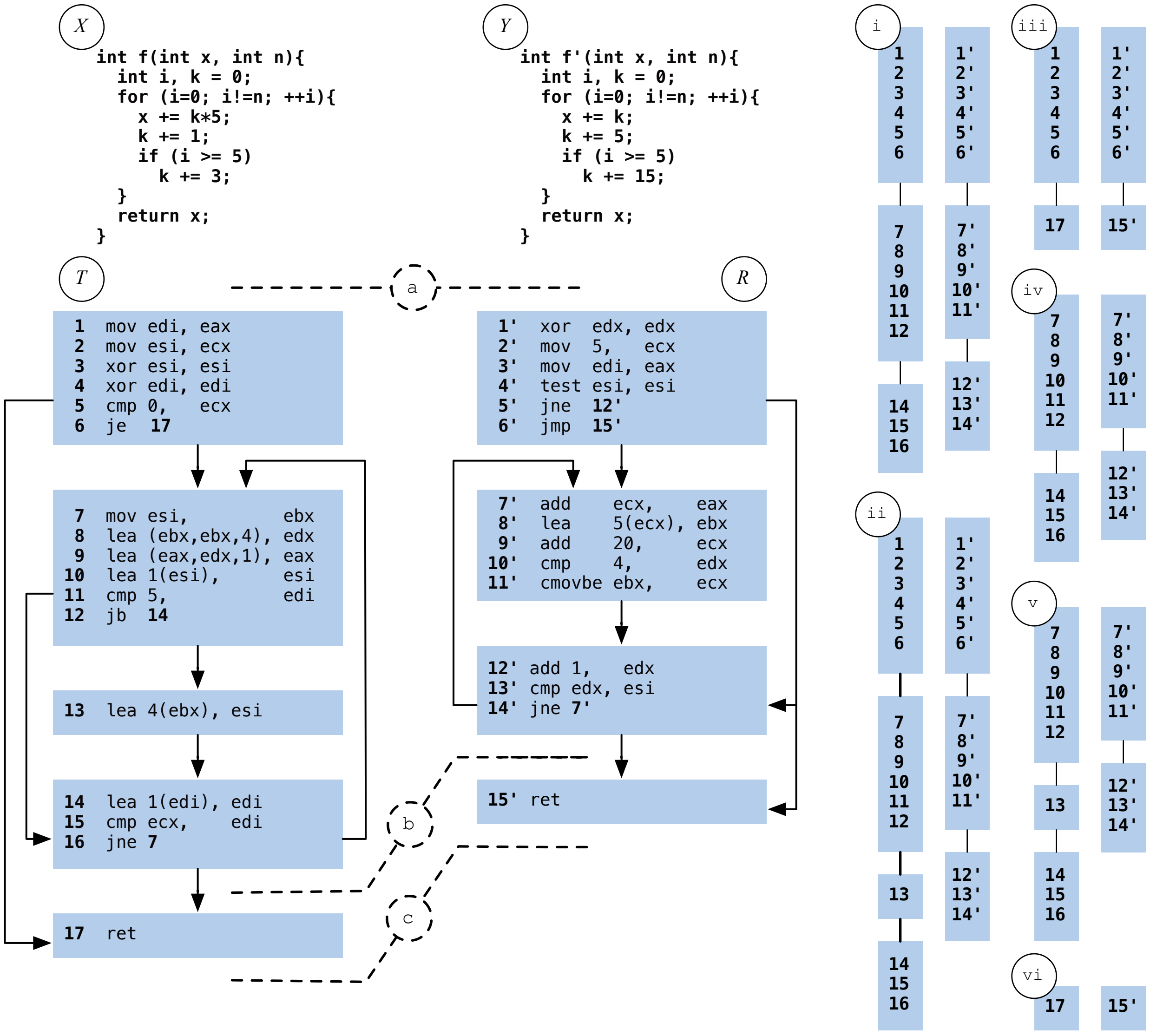
We begin by providing a detailed, but informal, worked example (Section 2), which is followed by a presentation of the equivalence checking algorithm (Section 3). Next we describe the implementation of DDEC (Section 4). Our evaluation (Section 5) shows DDEC is competitive with the state of the art in equivalence checking and presents the novel applications described above. Discussions of related work, limitations, and future work are included in Sections 6 and 7.

2. A Worked Example

Figure 1 shows two versions of a function taken from [36]. The straightforward implementation X is optimized using a strength reduction [2] to produce the code Y; corresponding x86 assembly codes T and R are shown beneath each source code function. We use primes ($'$) to represent program points and registers corresponding to the optimized code, which we call the *rewrite* R, and unprimed quantities for those corresponding to the unoptimized code, which we call the *target* T, throughout the paper. The generation of R also involves some low level compiler optimizations such as the use of an x86 conditional-move (instruction `11`$'$ of R) to eliminate a jump (instruction `12` of T). We are unaware of any fully automatic technique capable of verifying the equivalence of T and R. Nonetheless, our goal is to verify equivalence in the absence of source programs, manually written expert rules for equivalence, source code of the compiler(s) used, and compiler annotations—in other words, given only the two assembly programs and no other information. All of these features are necessary requirements to check equivalence at the assembly level when no knowledge is available about the relationship between the two codes. This situation arises, for example, in verifying the correctness of STOKE optimizations [33].

To verify that T and R are equivalent, we must confirm that whenever T and R begin execution in identical machine states and T runs to completion, R is guaranteed to terminate in the same machine state as T and with the same return value. In this example, which does not use memory, we limit our discussion of machine states to a valuation of the subset of hardware registers that the two codes use: `eax`, `ebx`, `ecx`, `esi`, and `edi` (our technique does not make this assumption in general and handles memory reads and writes soundly). We also assume that we know which registers are live on exit from the function, which in this case is just the return value `eax`. We stress that the following discussion takes place entirely at the assembly level and does not assume any information about the sources X or Y.

We use the well-known concept of a *cutpoint* [39] to decompose equivalence checking of two loops into manageable sub-parts. A cutpoint is a pair of program points, one in each program. Cutpoints are chosen to divide the loops into loop-free segments. The cutpoints in Figure 1, labeled `a`, `b`, and `c`, segment the programs as follows: 1) the code from `a` to `b` excluding the backedge of the loop, 2) the code that starts from `b`, takes the backedge once, and comes back to `b`, and 3) the code that starts from `b`, exits the loop, and terminates at `c`. Using these three cutpoints, we can produce an inductive proof of equivalence which proceeds as follows. Our goal is to show that the executions of T and R move together from one cutpoint to the next and that at each cutpoint, certain invariants are guaranteed to hold. The required invariants at `a` and `c` follow directly from the problem statement. At point `a`, we require that T and R agree on the initial machine states. Similarly at point `c`, we require that T and R agree on the return value stored in `eax`.



```
int f(int x, int n){
  int i, k = 0;
  for (i=0; i!=n; ++i){
    x += k*5;
    k += 1;
    if (i >= 5)
      k += 3;
  }
  return x;
}
```

```
int f'(int x, int n){
  int i, k = 0;
  for (i=0; i!=n; ++i){
    x += k;
    k += 5;
    if (i >= 5)
      k += 15;
  }
  return x;
}
```

```
 1  mov edi, eax
 2  mov esi, ecx
 3  xor esi, esi
 4  xor edi, edi
 5  cmp 0,   ecx
 6  je  17

 7  mov esi,          ebx
 8  lea (ebx,ebx,4), edx
 9  lea (eax,edx,1), eax
10  lea 1(esi),       esi
11  cmp 5,            edi
12  jb  14

13  lea 4(ebx), esi

14  lea 1(edi), edi
15  cmp ecx,    edi
16  jne 7

17  ret
```

```
1'  xor  edx, edx
2'  mov  5,   ecx
3'  mov  edi, eax
4'  test esi, esi
5'  jne  12'
6'  jmp  15'

7'  add    ecx,     eax
8'  lea    5(ecx), ebx
9'  add    20,      ecx
10' cmp    4,       edx
11' cmovbe ebx,     ecx

12' add 1,   edx
13' cmp edx, esi
14' jne 7'

15' ret
```

Figure 1. Equivalence checking for two possible compilations: (A) no optimizations applied either by hand or during compilation, (B) optimizations applied. Cut points (a,b,c) and corresponding paths (i-vi) are shown.

We now encounter two major difficulties in producing a proof of equivalence. The first problem we encounter is identifying the invariant that must hold at b. This invariant, I, is a relation between the machine states of T and R. In this paper, we consider invariants that consist of equality relationships between elements of the two machine states. Once we have identified the appropriate invariant I, an inductive proof would take the following form:

1. If T and R begin in a with identical machine states and transition immediately to c, they terminate with identical return values.
2. If T and R begin in a with identical machine states and transition to b, they both satisfy I.
3. If T and R begin in b satisfying I, and return to b, then I still holds.
4. If T and R begin in b satisfying I and transition to c, they terminate with identical return values.

However, this proof is still incomplete. It does not guarantee that if T makes a transition, then R makes the same transition: we do not for instance want T to transition from a to b while R transitions from b to c. The proof can be completed as follows.

Every transition between cutpoints is associated with some instructions of T and R: these are the instructions, or *code paths*, which need to be executed to move from one cutpoint to the next. For example, in moving from cutpoint `b` to `c`, T and R execute the instructions shown in code paths `vi` of Figure 1. A code path p_A of T *corresponds* to a code path p_B of R if they start and end at the same cutpoints and when T executes p_A then R executes p_B. Figure 1 shows a complete set of corresponding paths: `i` and `ii` are associated with transition `a-b`, `iii` is associated with `a-c`, `iv` and `v` are associated with `b-b` and `vi` is associated with `b-c`. We need to check that when T executes a code path then R can only execute the corresponding paths. Identifying these corresponding paths is a second major difficulty. The question of what invariants hold at `b` is crucial, as these must be strong enough to statically prove that the execution of R follows the corresponding paths of T and that the executions of both T and R proceed through the same sequence of cutpoints.

Our solution to both problems, identifying the equalities that hold at `b` and the corresponding paths of the two programs, is to analyze execution data. We identify corresponding paths by matching program traces for both loops on the same test inputs against the set of cutpoints. We observe the instructions executed by T and R in moving from one cutpoint to the next and label them as corresponding. For example, for a test case with initial state $\texttt{edi} = 0$ and $\texttt{esi} = \texttt{1}$, T begins its execution from `a` and executes instructions `1` to `16` to reach `b`. It then exits the loop and transitions from `b` to `c` by executing instruction `17`. R begins its execution from `a` and executes instructions $\texttt{1}'$ to $\texttt{14}'$ to reach `b`. It then executes instruction $\texttt{15}'$ to reach `c`. From this test case, we observe that code paths with instructions `1` to `16` correspond to instructions $\texttt{1}'$ to $\texttt{14}'$ (`i`) and `17` corresponds to $\texttt{15}'$ (`vi`).

The equality conditions at `b` can be determined by inserting instrumentation at `b` to record the values of the live program registers for both programs and observing the results. For a test input where $\texttt{edi} = 0$ and $\texttt{esi} = 2$, we obtain the following matrix where each row corresponds to the values in live registers when both programs pass through `b`:

$$\begin{bmatrix} \texttt{eax} & \texttt{esi} & \texttt{edi} & \texttt{ecx} & \texttt{eax}' & \texttt{ecx}' & \texttt{edx}' & \texttt{esi}' \\ 0 & 1 & 1 & 2 & 0 & 5 & 1 & 2 \\ 5 & 2 & 2 & 2 & 5 & 10 & 2 & 2 \end{bmatrix}$$

The first row says that when T reaches `b` for the first time the registers have the values shown in columns `eax` , `esi`, `edi`, and `ecx`; when R reaches `b` for the first time, the registers have values shown in $\texttt{eax}'$, $\texttt{ecx}'$, $\texttt{edx}'$, and $\texttt{esi}'$. The second row shows the values of registers when `b` is reached the second time, i.e., in the next iteration of T and R. Using standard linear algebra techniques, it is possible to extract the following relationships, which are sufficient candidates for the equalities which must hold at `b`: $\texttt{eax} = \texttt{eax}'$, $5{*}\texttt{esi} = \texttt{ecx}'$, $\texttt{edi} = \texttt{edx}'$, and $\texttt{ecx} = \texttt{esi}'$. The use of linear algebra on test data for invariant inference is well studied in software verification and the limitations are known. The process may generate spurious equality relationships [26], however these can be systematically eliminated using a theorem prover [34]. We note that this method assumes that equality relationships are sufficient to prove program equivalence and we do not, for example, consider invariants which contain inequalities. Previous work on translation validation [25, 36] makes the same assumption, which we find to be largely sufficient in practice (see Section 5).

Although it is possible that the tests used to generate these values do not produce sufficient coverage, and that either more corresponding paths exist, or spurious equality relationships are discovered at `b`, the consequence is simply that the proof will fail, and we will report that the two functions may be different. We may then reattempt the proof with more test cases, but a lack of test data cannot cause an unsound result. Barring this possibility, almost all of the limitations of this technique can be mapped to the restricted expressiveness of invariants. Better invariant generation techniques will only improve performance.

3. Algorithm

We now present a formal description of the algorithm sketched in Section 2. We assume we are given two functions T and R, each containing one natural loop and no function calls. We infer a candidate *simulation relation* consisting of cutpoints and linear equalities as invariants. Then we check whether the candidate is an actual simulation relation, and if so then we have a proof of equivalence. We consider functions containing multiple loops, calling other functions, and inference of non-linear invariants in Section 3.1.3.

As is standard, we make a distinction between T, the *target* or reference code, and R, the *rewrite* or proposed replacement for the target T. A program *state* consists of a valuation of registers, current stack frame, and memory. The memory consists of the whole virtual memory except the current stack frame. Generally, we will refer to the state at a program point; in such instances we will limit the state to only the live elements. We first define a suitable notion of equality:

Definition 1. *Target T is equivalent to rewrite R if for all possible states s, when execution of T is started from s and T terminates in final state s' without aborting then when rewrite R is executed from the initial state s, R terminates in state s' without aborting.*

This definition captures the fact that if T terminates on an input then so does R. Hence, this definition is richer than partial equivalence. A segmentation fault or a floating point exception are examples of aborting. The asymmetric notion of equality in Definition 1 seems necessary to validate several useful compiler optimizations. If the target T aborts on some inputs, optimizing compilers are free to use an R with any behavior, defined or undefined, on that input. The notion of equality we use is captured in the *verification conditions* (VCs) generated to symbolically compare the behavior of T and R. Since our VCs are in a rich logic, by modifying the VCs that are generated our approach can handle a wide range of specifications of equality, including requiring the target and rewrite to behave identically on every input or allowing the rewrite to differ from the target on any input that causes the target to abort.

Let t be a *code path*, that is, a sequence of instructions in T, r a code path in R, and C the pair $\langle t, r\rangle$. Abusing the notation of Hoare logic, we define proof obligations.

Definition 2. *For predicates P and Q and code paths t and r, a proof obligation $\{P\}\langle t, r\rangle\{Q\}$, states that if t starts execution from a state s_1, r start execution from a state s_2, $P(s_1, s_2)$ holds, and t terminates in a final state s_1' without aborting, then r does not abort and for all possible final states s_2' of r, $Q(s_1', s_2')$ holds.*

For example

$$\{\texttt{eax} = \texttt{eax}'\}\langle \texttt{mul } \$2 \texttt{ eax}, \texttt{shl } \$1 \texttt{ eax}\rangle\{\texttt{eax} = \texttt{eax}'\}$$

says if t and r begin with values of `eax` equal and t performs an unsigned multiplication by two and r shifts `eax` left by one bit then the resulting states agree on the value of `eax`. We want to generate sufficiently many proof obligations, such that if they are all satisfied then they constitute an inductive proof of equivalence of T and R.

3.1 Generating Proof Obligations

Each proof obligation $\{P\}C\{Q\}$ has two components: the code paths C and the predicates P and Q on the states of T and R. We discuss the generation of each component.

3.1.1 Generating Corresponding Paths

We generate corresponding paths t and r for proof obligations using *cutpoints* [39]. A cutpoint n is a pair of program points $\langle \eta_T, \eta_R\rangle$ where $\eta_T \in T$ and $\eta_R \in R$. We select cutpoints using the following heuristic: Choose pairs of program points where the corresponding memory states agree on the largest number of values. Cutpoints with higher agreement between the memory states of T and R have simpler invariants than cutpoints where the relationship between the two memory states is more involved.

The cutpoints are discovered using a brute force search that compares the execution data for pairs of program points. We create a candidate cutpoint for every program point pair $\langle \eta_T, \eta_R\rangle$. For every test σ, we find $m^\sigma_{\eta_T}$ (resp. $m^\sigma_{\eta_R}$), the number of times control flow passes through η_T (resp. η_R) when the execution of T (resp. R) starts in the state σ. If there do not exist constants a, b s.t. $\forall \sigma. m^\sigma_{\eta_T} = a m^\sigma_{\eta_R} + b$ then $\langle \eta_T, \eta_R\rangle$ is rejected as a candidate cutpoint. We also reject candidates for which the number of heap locations in the observed heap states for T and R at $\langle \eta_T, \eta_R\rangle$ is not a constant across all tests. Note that this operation is possible as we run only terminating tests and thus the memory footprint is bounded. For the remaining candidates, we assign them a penalty representing how different the observed heap states are for T and R at $\langle \eta_T, \eta_R\rangle$. We pick candidate cutpoints in increasing order of penalty until T and R are decomposed into loop free segments. Next, redundant candidates are removed so that the minimum number of program point pairs are selected as candidate cutpoints; a candidate n is redundant if the decomposition is still loop free after removing n. The choice of a minimum set of cutpoints may not be unique and we simply try multiple proofs, with one proof corresponding to every available choice. Multiple cutpoints can be associated with the same program point of T or R. For example, if a loop unrolling has been performed then there might be several cutpoints sharing the same program point of T.

If satisfactory cutpoints cannot be found then our technique fails, which highlights a limitation of our technique: we fail to prove equivalence for programs which differ from each other at an unbounded number of memory locations. For example, a loop fusion transformation or reordering an array traversal results in loops that cannot be proven equivalent using our method. Handling such loops requires quantified invariants, for which the current state of the art in invariant inference is less mature than for quantifier-free invariants. Generally, the techniques for quantified invariants require manually provided templates; the templates prevent the introduction of an unbounded number of quantified variables. Moreover, theorem provers are not as robust for quantified formulas as they are for quantifier-free formulas.

The set of cutpoints is called a *cutset* $\mathcal{S}$. For every pair of cutpoints n_1 and n_2 in a cutset $\mathcal{S}$, we define a *transition* $\tau \equiv n_1 \triangleright n_2$ from n_1 to n_2. This transition is *associated* with a set of static code paths $\mathcal{P}_1$ of T and $\mathcal{P}_2$ of R. $\mathcal{P}_1$ and $\mathcal{P}_2$ consists of code paths which go from n_1 to n_2 without passing through some other cutpoint $n_3 \in \mathcal{S}$. The instructions executed for a terminating and non-aborting execution of T and R can be represented by a sequence of transitions. For example, in Figure 1, let us denote by τ_1 the transition from `a` to `b` and τ_2 from `b` to `c`. Then the execution discussed in Section 2 represents the sequence of transitions $\tau_1\tau_2$. Code paths `i` and `ii` of Figure 1 are associated with τ_1 and `vi` is associated with τ_2. Now we define corresponding paths formally:

Definition 3. *Given a target T, a rewrite R, and a cutset $\mathcal{S}$, a code path t of T corresponds to a code path r of R if they are associated with the same transition τ in $\mathcal{S}$ and for some execution $\tau_1, \ldots, \tau, \ldots, \tau_m$, for the transition τ, if T executes the code path t then R executes r.*

Now we discuss our data-driven algorithm for generation of corresponding paths. T and R are run on a test input and we record the trace of instructions executed in both functions. We analyze this pair of traces to build a set $\mathcal{C}$ of corresponding paths. The algorithm walks over the traces in parallel until a cutpoint is reached in both. The pair of paths taken from the previous pair of cutpoints is added to $\mathcal{C}$. This process is repeated until we reach the end of the traces. We then run the two functions on another input and repeat the analysis. If test coverage is sufficient, $\mathcal{C}$ will contain all corresponding paths.

We add additional proof obligations to ensure that we have not missed any corresponding paths. For static paths in T between cutpoints that no test executes, we add an arbitrary path from R between the same cutpoints as a corresponding path in $\mathcal{C}$. For a path p of T associated with a transition τ, we add the verification condition that if the target executes p then the rewrite can only execute the paths which correspond to p in $\mathcal{C}$. If these proof obligations fail then this means that our data is insufficient and there are more corresponding paths to be discovered.

3.1.2 Generating Invariants

Following Necula [25], we consider invariants that are equalities over *features*, which are registers, stack locations and a finite set Δ of heap locations. Also as in [25], we replace reads from and writes to stack locations by reads and writes to named temporaries.

We perform a liveness analysis over T and R and for each cutpoint (η_1, η_2) obtain the live features (see Section 4). Because x86 has sub-registers, these features can be of different bit-widths. For example, it might be that only one byte of a 64-bit register is live. We create a set of matrices, one for each cutpoint (η_1, η_2), whose columns are the live features at η_1 and η_2. If the bit-width of the longest live feature is x bits, we create one column for every x bit live feature. For every live feature of fewer than x bits, we create two columns: one with the feature's value zero-extended to x bits and the other with the value sign-extended. The sign- and zero-extensions are needed because x86 instructions implicitly perform these operations on registers of different bit-widths.

We instrument the functions to record the values of the features at the cutpoints; a snapshot of the state at one cutpoint corresponds to a single matrix row. If the cutpoint (η_1, η_2) is executed m times then we have m rows in the matrix for (η_1, η_2). Note that there are no negative values in the matrices: negative numbers become large positive numbers.

We use elementary linear algebra to compute the linear equality relationships between features. For the matrix associated with each cutpoint, we compute its *nullspace* or *kernel*. Every vector of the nullspace corresponds to an equality relationship between features at that cutpoint for all test inputs. We take a conjunction of equalities generated by all vectors in the basis of the nullspace and return the resulting predicate as a candidate invariant for our candidate simulation relation. For example, if our matrix has three features and two rows:

$$\begin{bmatrix} \texttt{eax} & \texttt{ebx} & \texttt{eax}' \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

then one possible basis of the nullspace consists of the vectors $[1, -1, 0]$ and $[0, 1, -1]$. These vectors correspond to the equalities $\texttt{eax} * (1) + \texttt{ebx} * (-1) + \texttt{eax}' * 0 = 0$ and $\texttt{eax} * 0 + \texttt{ebx} * (1) + \texttt{eax}' * (-1) = 0$. Hence the candidate invariant is $\texttt{eax} = \texttt{ebx} \wedge \texttt{ebx} = \texttt{eax}'$. Note that the heap locations, which are not included in features, are implicitly constrained to have identical values for T and R. Hence, the above invariant includes an implicit equality $H = H'$, that is, for all heap addresses, at the cutpoint, T and R have identical values. We use a nullspace algorithm for rational matrices, which ensures that any discovered equality is exact. The cutpoints are labeled with these equalities. One desirable feature of nullspaces is that no sound equality relationship is missed. It can produce spurious equality relationships (for lack of sufficient data) but if an equality holds statically then it will be generated [34]. Intuitively, this is because every possible equality is contained in the candidate invariant unless there is a test that violates it. In the extreme, when we have zero states in our data then the candidate invariant consists of every possible equality between the features, and hence it also includes the equalities present in the true invariants. We generalize the results in [34], which uses nullspaces to compute invariants for a single program, to features:

Lemma 1. *If x is a set of features at a cutpoint n, and $\mathcal{I}(x)$ is the strongest invariant at n that holds statically and is expressible by conjunctions of linear equalities, then the candidate invariant $I(x)$ obtained by computing the nullspace of test data is a sound under-approximation of $\mathcal{I}$, that is, $I \Rightarrow \mathcal{I}$.*

Before we discuss how candidate invariants are checked to see if they are in fact invariants, we briefly digress to discuss how our approach extends to more complex function bodies.

3.1.3 Extensions

Our approach easily generalizes to functions with multiple natural loops, including nested loops. We identify the cutset of T and R and identify cutpoints and transitions between two cutpoints if there is a static path from one cutpoint to the other. As is standard, we want every loop to contain at least one cutpoint of the cutset. The algorithm described above for generating proof obligations remains unchanged: run tests, generate corresponding paths, and generate equality relationships at cutpoints.

We can also extend our approach to handle loops that call other functions. The function call is a cutpoint requiring the invariant that the value of arguments and memory is same across T and R and after the call we can assume that the memory and return values are equal in the proof obligations. We generate additional obligations to check that the order of function calls is preserved.

Our approach easily extends to generate non-linear equalities of a given degree d for invariants using ideas from invariant inference: We simply create a new feature for every monomial up to the degree d from the existing features [26, 34]. Say if eax and ebx are features and the chosen degree is 2 then we add $\texttt{eax}^2$, $\texttt{ebx}^2$, and $\texttt{eax} * \texttt{ebx}$ to the existing set of features. Now the nullspace yields linear equalities over the extended features which represent polynomial equalities over the original features. Since non-linear invariants of degree d can represent disjunctions of d linear equalities (e.g., $x^2 = y^2 \equiv x = y \vee x = -y$), the expressiveness of the invariants we can infer includes boolean combinations of linear equalities with a given number of disjunctions.

3.2 Checking Proof Obligations

For every transition τ between cutpoints n_1 and n_2 where the candidate invariant at n_1 is equality relationship P, the candidate invariant at n_2 is equality relationship Q, and C is a pair of corresponding paths associated with τ, we construct a proof obligation $\{P\}C\{Q\}$. As noted in Section 2, we also generate additional verification conditions or VCs to check that all pairs of corresponding paths are covered. Once the VCs have been obtained, they can be sent to an off-the-shelf theorem prover. These queries are in the quantifier-free theory of bit-vectors. More details about generating VCs can be found in Section 4.6. The proof obligations $\{P\}C\{Q\}$ are of three types:

- $\{E\}C\{Q\}$, where E represents exact equivalence between states. Here the corresponding paths are code paths from the start of T and R to a cutpoint and Q is the equality relationship associated with that cutpoint.
- $\{P\}C\{Q\}$, where for $C = \langle t, r\rangle$, t and r start at a cutpoint n_1 and end at a cutpoint n_2. P and Q are the equality relationships at n_1 and n_2.
- $\{P\}C\{F\}$, where the corresponding paths start at a cutpoint and end at a return statement of T and R. Here F expresses that the return values are equal and the memory states are equivalent.

If proof obligation $\{E\}C\{Q\}$ or $\{P\}C\{Q\}$ fails then we can obtain a counter-example from the decision procedure and follow the approach of [34]: If the counter-example violates some equality of Q then we incorporate the data from the counter-example in the appropriate matrix. Next, we recompute the nullspace to obtain new equality relationships satisfying both the data and the counter-example. If the nullspace is just the null vector then we return the trivial invariant *true*. This process systematically guides our data-driven algorithm to generate equality relationships that are actual invariants. It also removes some artifacts of test cases: for example, in all tests some pointer p is assigned the same address a then we can infer a spurious equality relationship that $p = a$. A counter-example can remove this spurious equality. However, if the counter-example satisfies Q then the proof fails; this can happen because r might abort on some state for which t does not.

Note that [34] contains a completeness theorem, which is possible because it deals with a restricted language with no heap and has no specification to verify. Since we handle memory, the possibility of aborting, and need to prove equivalence, we are only able to prove a soundness theorem:

Theorem 1. *If all VCs are proven then the target is equivalent to the rewrite and the cutset and the invariants together constitute a simulation relation.*

4. Implementation

DDEC is a prototype implementation of the algorithm discussed in Section 3. We discuss the most important and interesting features of the implementation below.

4.1 Liveness Computation

Several components of DDEC require the live variables at a program point. For the most part, liveness is computed using a standard dataflow algorithm, but we note the following modification for the x86 architecture. As mentioned previously, for 64-bit x86 some general purpose registers have subregisters. For example, `dil`, `di`, and `edi` refer to the lowest 8, 16, and 32-bits of `rdi`, respectively:

$$\texttt{dil} \subset \texttt{di} \subset \texttt{edi} \subset \texttt{rdi}$$

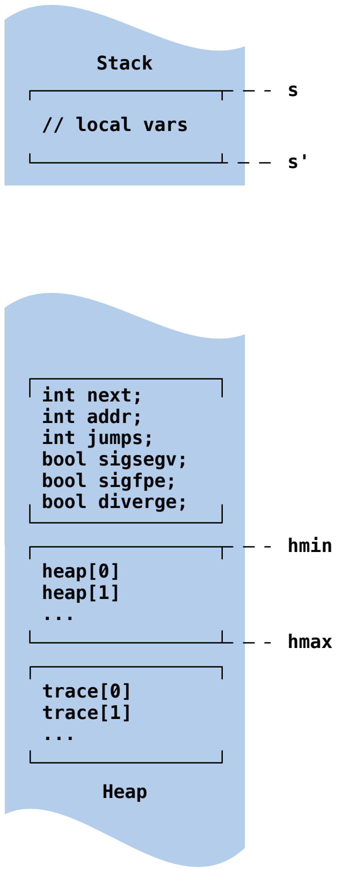
To account for this register aliasing, when calculating the read set of an instruction we include the registers it reads, along with all subsets of those registers. An instruction that reads `di`, for example, also reads `dil`. Similarly, when calculating the write set of an instruction we include the registers it writes, along with all super- and subsets of those registers. Some instructions also produce undefined register values, which are treated as killing any live values.

4.2 Testcase Generation

For programs that deal exclusively with stack and register values, DDEC is able to produce tests automatically. Using the control flow graph of the target (recall the distinction between the target and rewrite introduced in Section 3), DDEC identifies the set of live-in registers and generates random values as necessary. For programs that dereference heap locations, DDEC requires a user-defined environment in which to execute the target. For large software projects, we expect that a representative set of whole-program tests will exist, and DDEC may simply observe the behavior of the code during normal program execution. When no such inputs exist, a user must provide a small unit test harness which will establish whatever heap structures are necessary to enable the functions to execute correctly.

4.3 Target Tracing

To observe the values of live program variables at cut-points, we rely on the ability to instrument the target and observe those values dynamically as they appear under representative inputs. This instrumentation is performed using a custom light-weight JIT assembler for 64-bit x86; the `trace()` function is shown in Figure 2. Prior to execution, an array



```
void trace(Code c){
  EMIT(next = 0);
  EMIT(save_callee_save_state());

  foreach (Instr i : c){
    if (mem_deref(i))
      EMIT(addr = deref_addr(i));

    EMIT(record(i, next++));
    EMIT(i);
    EMIT(record(i, next++));
  }
}


void sandbox(Code c){
  EMIT(next = 0);

  foreach (Instr i : c){
    if (mem_deref(i))
      EMIT(sandbox_addr(i));

    EMIT(record(i, next++));

    if (backedge_jump(i))
      EMIT(sandbox_jump(i));
    else if (is_return(i))
      EMIT(sandbox_return());
    else if (is_mem_deref(i))
      EMIT(sandbox_deref(i));
    else
      EMIT(i);

    EMIT(record(i, next++));
  }
}


void error(bool bit){
  bit = true;
  restore_callee_saved_state();
  ret;
}
```

```
void sandbox_addr(Instr i){
  addr = deref_addr(i);
  if (addr >= s)
    error(sigsegv);
  if (addr >= s')
    return;
  addr -= hmin;
  if (addr<0 || addr>hmax)
    error(sigsegv);
  addr -= &heap[0];
}


void record(Instr i, int next){
  foreach (GpReg r : gp)
    trace[next].gp[r] = val(r);
  foreach (XmmReg x : xmm)
    trace[next].xmm[x] = val(x);
  foreach (Flag f : eflags)
    trace[next].ef[f] = val(f);
  if (mem_deref(i)) {
    trace[next].addr = addr;
    trace[next].val = *addr;
  }
}


void sandbox_jump(Instr jump){
  if (++jumps == max)
    error(diverge);
  jump;
}


void sandbox_return(Instr ret){
  if (!callee_save_restored())
    error(segv);
  ret;
}


void sandbox_deref(Instr i){
  i.set_deref_addr(addr);
  i;
}
```

Figure 2. Implementation of instrumentation for target and rewrite tracing using a JIT assembler. Targets are assumed safe and instrumented using the `trace()` function. Rewrites are instrumented using the `sandbox()` function which is parameterized by values observed during the execution of the target.

of `trace` states are allocated on the heap. For every instruction executed by the target, we emit a call to the `record()` function to copy the majority of the hardware state (general purpose, SSE, and condition registers) as well as values read from or written to the heap, to the back of the array. Array bounds are tracked (not shown), and in the event of overflow additional capacity is added as necessary. When the target function completes execution, the trace array holds a complete record of the values manipulated by the target in both registers and memory.

Because we already assume that users supply tests for target functions that access data structures in memory, in our prototype we have also assumed that the target is safe and will not crash the machine or corrupt DDEC's state on any of the supplied inputs. This assumption held for the experiments in this paper, however for a production tool it would be necessary to isolate the target from the tool using well-known but more involved techniques (e.g., emulation).

4.4 Rewrite Tracing

The correctness assumptions that we make for the target do not hold for all the rewrites in our experiments. For example, some candidate rewrites can and do dereference invalid memory locations (see Section 5.4). Thus, our prototype instruments rewrites using the heavier-weight `sandbox()` function (see Figure 2). We can, however, take advantage of information observed in tracing the target to simplify the tracing of the rewrite.

From the execution of the target, we obtain the maximum stack size used along with the minimum and maximum heap addresses that were dereferenced. Using these values, we define stack and heap sandboxes to guard the execution of the rewrite. Specifically, we identify the stack pointer, `s`, which defines the upper bound on the frame used by the rewrite, and from this value identify s', a location which defines a frame no larger than the one used by the target. Similarly, we allocate a heap sandbox array large enough to contain each heap dereference performed by the target without aliasing containing values which are initialized to the live-in memory values dereferenced by the target. Memory dereferences are guarded by the `sandbox_addr()` function, which preserves stack accesses inside the bounds of the stack sandbox, redirects valid heap accesses to the heap sandbox, and traps all other accesses and instead produces a safe premature termination. If necessary, both sandboxes may be scaled by a constant user-defined factor to allow for the possibility of a rewrite with greater memory requirements than the target.

In addition to sandboxing memory accesses, several other behaviors need to be checked to protect DDEC from undefined behavior. First, there is the possibility that the rewrite will go into an infinite loop. This behavior is guarded by a call to `sandbox_jump()`, which counts the number of times that a backedge is taken and causes a premature termination if a bound calculated from the number of backwards jumps taken in the target execution is exceeded. Second, we must guarantee that return instructions take place only after certain invariants specific to the x86 application binary interface have been restored. This property is guaranteed by a call to `sandbox_return()`, which checks that the value of callee-saved registers are restored to the state they held when the rewrite began executing.

4.5 Invariant Generation

Invariants are computed using the `nullspace` function of the Integer Matrix Library [5] which is specialized for computing the nullspace of integer and rational matrices using p-adic arithmetic. DDEC uses sixteen random tests to generate data for invariant computation. For the experiments in this paper, these tests were sufficient for obtaining sound invariants and all necessary corresponding paths. For a production system, a larger number of tests would likely be necessary. However at under 2 ms for each null space computation, we do not expect that this computation would be a bottleneck.

4.6 VC Generation

Proof obligations are discharged using Z3 [7]. Because Z3 does not currently provide support for floating point operations, DDEC's ability to reason about such instructions is limited as well. However, Z3 does have a complete algorithm for real numbers and one might be tempted to use this capability to reason about floating point programs. Unfortunately, floating point semantics are not over-approximated by reals—a simple example is the non-associativity of floating point addition. Hence, sound optimizations using real semantics may be unsound using floating point semantics.

We manually encode x86 instructions as Z3 formulas, primarily due to lack of access to formal specification of x86 instructions at the hardware level. For example, some x86 instructions, such as `popcnt`, are informally described by Intel as loops. Wherever possible, we deferred to corresponding loop-free implementations given in A Hacker's Delight [40]. Our formulas precisely model the complexity of the x86 instruction set, which in some cases is quite sophisticated (consider the `crc` instruction, which considers bit-vectors as polynomials in $\mathbb{Z}_2$ and performs a polynomial division). Formula correctness is a necessary prerequisite for proof correctness, and unfortunately, testing is the only available option for ensuring this property. Each formula was tested extensively against hardware semi-automatically to check correctness. In the process we rediscovered known instances in which the x86 instruction set deviates from its specification [12]. For such instances, our formulas encode a sound over-approximation of the observed hardware behavior and the specification.

We generate VCs in the quantifier-free theory of bit-vector arithmetic. Z3 is complete for this theory [41] and is able to soundly analyze the effect of x86 instructions on the machine state with bit-wise precision. When generating constraints, registers are modeled, depending on bit-width, as between 8- and 128-bit bit-vectors. Memory is modeled as two vectors: one of 64-bit addresses and one of corresponding 8-bit values (x86 is byte addressable). For 32-bit x86, addresses are restricted to 32 bits.

DDEC translates proof obligations $\{P\}\langle t, r\rangle\{Q\}$, where P and Q are predicates over registers (the invariants can be more general as described in Section 3.1.2), to a VC as follows. DDEC first asserts the constraint P. It then iterates over the instructions in t, written in SSA form, and for each instruction asserts a constraint which encodes the transformation the instruction induces on the current hardware state. These constraints are chained together to produce a constraint on the final state of live outputs with respect to t. Analogous constraints are asserted for r. In our implementation, operators that are very expensive for Z3 to analyze, such as bit vector multiplication and division, are replaced by uninterpreted functions constrained by some common axioms. Finally, for all pairs of memory accesses at addresses $\texttt{addr}_1$ and $\texttt{addr}_2$, DDEC asserts additional constraints relating their values: $\texttt{addr}_1 = \texttt{addr}_2 \Rightarrow \texttt{val}_1 = \texttt{val}_2$. These aliasing constraints grow quadratically in the number of addresses, though dependency analysis can simplify the constraints in many cases.

Using these constraints, DDEC constructs a Z3 query which asks whether there exists an initial state satisfying P that causes the two code paths to produce values for live outputs which either violate Q or result in different memory states. If the answer is no, then the obligation is discharged, otherwise the prover produces a counter example, and DDEC fails to verify equivalence. If every VC is discharged successfully, then DDEC has proven that the two functions are equivalent.

Although the implementation described above is correct, it is overly conservative with respect to stack accesses. Specifically, an access to a spill slot [2] will appear indistinguishable from a memory dereference, and Z3 will discover a counter-example in which the input addresses to t and r alias with respect to that slot. As a result, any sound optimized code which eliminates stack traffic will be rejected. We address this issue by borrowing an idea from Necula [25], who solves this problem by replacing spill slots with temporary registers that eliminate the possibility of aliasing between addresses passed as arguments and the stack frame. For well-studied compilers such as gcc and CompCert, simple pattern matching heuristics are known to be well-suited to this task [25]. For example, for code compiled with gcc, all stack accesses are at constant offsets from the register *rbp*. We can create a new register for *rbp-8*, *rbp-16*, and so on and model loads and stores from the stack frame by loads and stores from these temporaries. Modeling spill slots in this way could result in unsound results if an input address is used to form an offset into the current stack frame. However, this behavior is undefined even in the high-level languages that in theory permit it (e.g., C) and, to the best of our knowledge, existing optimizing compilers are also unsound for such programs.

Given that we have gone to the effort to construct a faithful symbolic encoding of the x86 instruction set, the reader might wonder why we do not simply obtain the simulation relation statically—what does data add? The answer is that inference is harder than checking. Consider an example where we compute the dot product of two 32-bit arrays, where the multiplication of two 32-bit unsigned numbers is used to produce a 64-bit result. Say the target uses Karatsuba's trick [19] and performs three 32-bit *signed* multiplications to obtain a 64-bit result, whereas the rewrite uses a special x86 instruction that performs an *unsigned* multiplication of two 32-bit numbers and directly produces a 64-bit result. A static analysis that attempts to discover the relationship between Karatsuba's trick and the special x86 instruction has a very large search space containing many other plausible proof strategies to sift through, and in fact we know of no inference technique that can reliably perform such reasoning. In contrast, with DDEC the equality simply manifests itself in the data, and knowing that this specific equality is what needs to be checked narrows the search space to the point that off-the-shelf solvers can verify this fact automatically.

5. Experiments

We summarize our experiments with DDEC on a number of benchmarks drawn from both the literature and from existing compiler test suites. In doing so, we demonstrate the relationship between DDEC and the state of the art in translation validation, the use of DDEC in the optimization of a production codebase, and show how DDEC enables the design of novel applications using existing compiler toolchains. Specifically, we use DDEC to combine the correctness guarantees of CompCert with the performance properties of gcc -O2, and to extend the applicability of a binary superoptimizer to functions containing loops.

Program	LOC	#Loop	Run-time
lerner1a1b	29/26	1	19.39s
lerner3b3c	32/32	2	102.89s
unroll	13/20	1	75.04s
off-by-one	15/14	1	0.13s

Table 1. Performance results for micro-benchmarks. LOC shows lines of assembly of target/rewrite pair.

Program	LOC	#Loop	Run-time	Speedup	Test
lerner1a	39/17	1	12.94s	none	4
lerner3b	43/27	2	53.72s	1.49x	5
bansal	19/10	1	9.89s	1.02x	2
chomp	30/22	1	11.00s	none	3
fannkuch	30/23	1	17.03s	1.11x	4
knucleotide	28/21	1	6.56s	none	3
lists	21/17	1	1.40s	none	3
nsievebits*	70/38	2	36.44s	1.20x	5
nsieve*	44/31	2	166.31s	1.06x	5
qsort*	64/56	3	140.87s	1.05x	7
sha1*	79/28	1	12888.87s	1.07x	8

Table 2. Performance results for CompCert and gcc equivalence checking for the integer subset of the CompCert benchmarks and the benchmarks in this paper. LOC shows lines of assembly code for the binaries generated by CompCert/gcc. Test is the maximum number of random tests required to generate a proof. A star indicates that a unification of jump instructions was required.

Program	LOC	#Loop	Run-time	Speedup O0/O3
Bansal	9/6	1	44s/5492.75s	1.58x/1.04x
SAXPY	9/9	1	211s/0.62s	9.22x/1.48x

Table 3. Performance results for gcc and STOKE+DDEC equivalence checking for the loop failure benchmarks in [33]. LOC shows lines of assembly code for the binaries generated by STOKE/STOKE+DDEC. Run-times for search/verification are shown along with speedups over gcc -O0/O3.

Experiments were performed on a 3.40 GHz Intel Core i7-2600 CPU with 16 GB of RAM. For each experiment we report the number of assembly instructions for target and rewrite, the number of loops contained in each function, the run-time required for DDEC to verify equivalence, and where applicable the observed performance improvement between target and rewrite. Run-times for all experiments were tractable, varying from under a second to several hours, depending on the complexity of the constraints. Memory usage was not an issue and did not exceed 500Mb.

It is worth noting that many of the invariants discovered by DDEC, such as $4 * \mathtt{eax} = \mathtt{edx}' + 3$ and $10 * \mathtt{eax} + \mathtt{edx} = \mathtt{ecx}'$, were non-trivial. In all cases the run-time required to run tests, infer equality relationships and generate proof obligations was minimal (a few milliseconds) compared to the run-time required to discharge the VCs. Essentially all of DDEC's run-time was spent in Z3 queries.

```
// Target:
while( p != null ) { p=p->next; ... }

// Rewrite (Unrolled once):
while( p != null ) {
  p=p->next; ...
  if ( p != null ) { p=p->next; ... }
}
```

Figure 3. Abstracted codes (target and rewrite) for `unroll` of Table 1.

```
// Target:
for ( i = len - 1;     i >= 0; --i ) { ... }

// Rewrite (with off-by-one error):
for ( i = len /*BUG*/; i >= 0; --i ) { ... }
```

Figure 4. Abstracted codes (target and rewrite) for `off-by-one` of Table 1.

5.1 Micro-benchmarks

In this section, we attempt to provide a detailed comparison between DDEC and techniques based on equality saturation [18, 36]. Equality saturation [36] is a state of the art technique for verifying compiler optimizations which relies on expert-written equality rules, such as "multiplication by two is equivalent to a bit shift". Beginning from both the target and the rewrite, the repeated application of these rules is used to attempt to identify a common representation of both functions, the existence of which implies equality. Unsurprisingly, the technique is precisely as good as the expert rules that it has at its disposal. Missing rules correspond to optimizations for which equivalence cannot be proven. Furthermore, identifying such rules for a CISC architecture such as x86 is a daunting task. As a result, the application of equality saturation has been limited to the verification of optimizations in several intermediate RTL languages.

We compare DDEC to equality saturation using the two motivating examples described in the original paper [36], the first of which appears in Section 2; these correspond to the first two rows in Table 1. Each example consists of two pairs of programs demonstrating strength reductions. We compile each pair with `gcc -O0` to both transform the functions into a format that DDEC accepts and to preserve the structure of the original optimizations as best as possible. DDEC successfully proves the equivalence of the resulting binaries.

Equality saturation, in its current form, is unable to handle important optimizations such as loop unrolling [36]. The `unroll` benchmark in Table 1 is an equivalence checking task involving loop unrolling. We take a program which walks over a linked list and unroll it to obtain a new program that walks over two elements in every iteration (Figure 3). Hence, we are able to handle optimizations that equality saturation cannot even in the presence of an extensive set of expert written rules.

When given two programs that are semantically different, the process of equality saturation is not guaranteed to terminate. It simply continues applying rules indefinitely in an attempt to prove that the two programs are equal. When given two semantically different programs, DDEC always terminates with a counter-example (Z3 is complete for the VCs we use). Note that counter-examples are also possible even for equivalent programs because DDEC is incomplete.

Counter-examples from a failed proof can be used to explain why DDEC believes that a target/rewrite pair are not equivalent. To demonstrate this ability, we take a program which walks backwards over an array and introduce an off-by-one error (Figure 4). We compile both programs with `gcc -O0` and use the correct program as the target and the buggy version as the rewrite. When we ask DDEC to prove the equivalence of the two programs, DDEC fails and terminates with a counter-example (`off-by-one` in Table 1). Hence, DDEC can be used for finding equivalence bugs. However, in general, it is difficult to map the counter-example at the assembly level to the source code, and especially in the presence of compiler optimizations. A version of DDEC that works on source code and finds equivalence bugs is left as future work.

5.2 Full System Benchmark

One criticism of DDEC is that it is not purely static: it requires tests. However, we believe that for many systems existing regression tests should suffice. To validate this belief, we perform a case study with OpenSSL. The core computation routine of OpenSSL is big number multiplication: a loop which multiplies n-bit numbers stored as arrays. OpenSSL includes performance tests which use this multiplication loop extensively. For example, the RSA performance test included with OpenSSL executes the core more than fifteen million times.

We instrument OpenSSL to obtain these tests and select sixteen tests at random from the large number performed. We compile the core using `gcc -O0` and `gcc -O3` and use the tests to drive DDEC, which is then able to prove the equivalence in about two hours. We also find that the maximum time taken by any individual VC is ten minutes. Since checking proof obligations is an embarrassingly parallel task, if we assign a single cpu for every obligation then the proof can be obtained in ten minutes. When we measure the sensitivity of DDEC to the number of tests for this example, we find that 6-8 randomly selected tests are sufficient to obtain a proof.

Hence, for verifying the optimizations of kernels, it seems possible to obtain sufficient tests from existing test suites to drive verification. The tests included with programs might not exercise all parts of the code but they will exercise the performance-critical parts well and hence we believe DDEC can be successfully applied to the performance-critical kernels. Being data-driven, our technique will fail to prove interesting optimizations performed on dead code or the part of code exercised rarely by tests; a static equivalence checking engine for x86, if it existed, could have succeeded here.

5.3 CompCert

COMPCERT is a certified compiler for a subset of the C programming language: it is guaranteed to produce binaries that are semantically equivalent to the input source code. COMPCERT 1.12, the current version, does not perform loop optimizations and its compilation model does not fit well with CISC architectures such as x86, due to their "paucity of [general purpose] registers" [21].

We use DDEC to verify the equivalence of binaries generated by COMPCERT and `gcc -O2 -m32`. The `-m32` flag is necessary for compatibility with COMPCERT, which only produces 32-bit x86 binaries. Furthermore, optimization is restricted to `-O2` because, for these benchmarks, higher optimization levels produce only syntactic differences that do not affect DDEC. In doing so, we are able to extend the correctness guarantees made by COMPCERT to the more performant but uncertified code generated by `gcc`.

Performance data is shown in Table 2 for the integer subset of the COMPCERT compiler test suite (the benchmarks in the `test/c` directory of the COMPCERT 1.12 download) and several of the benchmarks described in this section. For the COMPCERT benchmarks, we profile each program and report results for the one loop (possibly containing other loops) that dominates execution time. The results are encouraging; DDEC is able to prove equivalence in all cases. We note, however, that DDEC did encounter difficulty with some of the benchmarks due to the restrictive logic that it uses. The problem is illustrated by the following example:

Target:

```
if(0<n) {for (i=0; i<n; i++);} return i;
```

Rewrite:

```
if(0<n) {for (i=0; i!=n; i++);} return i;
```

To prove the equivalence of these two functions, DDEC requires the inductive invariant $i = i' \wedge n = n' \wedge i \leq n$. Crucially, $i \leq n$ is inexpressible with equalities. For the final four benchmarks in Table 2 (the ones marked with a star), DDEC reported failure due to `gcc` replacing an inequality by a dis-equality. For these benchmarks it was necessary to manually make the test predicates identical before performing a successful equality verification. DDEC can also be combined with an abstract interpreter over binaries [31, 37]: these are capable of finding invariants over state elements of a single program (such as $i \leq n$). It is also straightforward to extend DDEC to recognize this common special case, but a solution that incorporates inequality relationships between state elements of two programs in a general way is challenging and we leave it for future work.

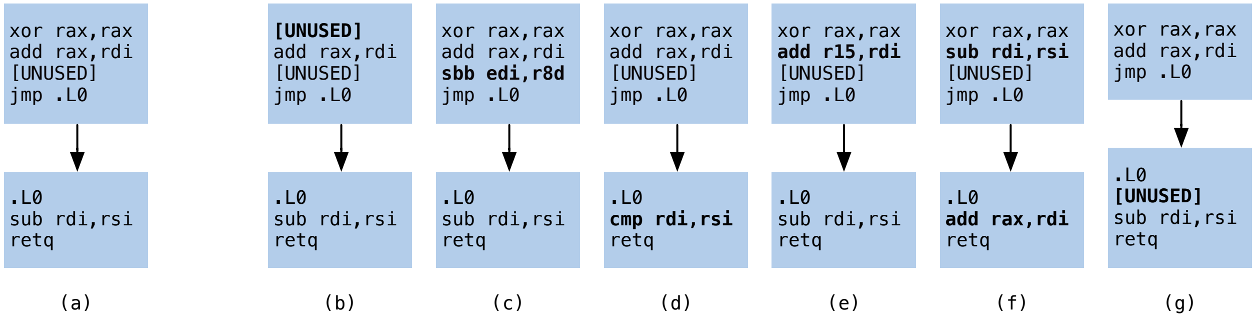
Finally, we note that the long verification time required for `sha1` is due to the large numeric constants used by this cryptographic computation that lead to poor performance in the underlying Z3 theorem prover. We confirmed that if these constants are replaced by small integers the proof obligations are discharged in a few minutes.

5.4 STOKE

STOKE [33] is an x86 binary superoptimizer based on the principle that program optimization can be cast as a stochastic search problem. Given a suitable cost function and run for long enough (which may be an extremely long time), STOKE is guaranteed to produce a code with the lowest cost [33]. STOKE performs binary optimization by executing up to billions of small random modifications to a program. For each modification, STOKE evaluates a cost function representing a combination of correctness and performance metrics. STOKE accepts all modifications that decrease the value of this function, and with some probability also accepts modifications that increase the value. Although most of the programs that STOKE considers are ill-formed, the sheer volume of programs that it evaluates leads it in practice to discover correct optimizations. STOKE runs for a user-defined amount of time and returns the lowest cost program that it can prove equivalent to the original program.

Previous work on STOKE demonstrated promising results for loop-free kernels. Beginning from code compiled using `llvm -O0`, STOKE produces programs that outperform code produced by `gcc -O3` and in some cases outperforms handwritten assembly. Unfortunately, STOKE's application to many interesting high-performance kernels is limited by its inability to reason about equivalence of codes containing loops. We address this limitation by extending STOKE's set of program transformations to include moves which enable loop optimizations and replacing STOKE's equivalence checker by DDEC. Our version of STOKE is able to produce optimizations for the loop benchmarks that could not be fully optimized in [33].

STOKE's underlying program representation is a constant length sequence of 64-bit x86 instructions. STOKE uses a special [UNUSED] token in place of a valid x86 instruction to represent programs that are shorter than this length. STOKE explores candidate rewrites using the following transformations. Instruction moves replace a random instruction with either the [UNUSED] token or with a completely different random instruction. Operand and opcode moves randomly replace the value of a random operand or opcode respectively. And finally, swap moves interchange two instructions at random. STOKE's program transformations are all intra-basic block, and it does not allow moves which modify control flow structure. Our implementation of STOKE remedies these shortcomings by introducing an inter-basic block swap move (Figure 5f) and a basic block resizing move (Figure 5g), which changes the number of instructions allowed inside a basic block. These simple exten-



```
xor rax,rax
add rax,rdi
[UNUSED]
jmp .L0

.L0
sub rdi,rsi
retq
```

(a)

```
[UNUSED]
add rax,rdi
[UNUSED]
jmp .L0

.L0
sub rdi,rsi
retq
```

(b)

```
xor rax,rax
add rax,rdi
sbb edi,r8d
jmp .L0

.L0
sub rdi,rsi
retq
```

(c)

```
xor rax,rax
add rax,rdi
[UNUSED]
jmp .L0

.L0
cmp rdi,rsi
retq
```

(d)

```
xor rax,rax
add r15,rdi
[UNUSED]
jmp .L0

.L0
sub rdi,rsi
retq
```

(e)

```
xor rax,rax
sub rdi,rsi
[UNUSED]
jmp .L0

.L0
add rax,rdi
retq
```

(f)

```
xor rax,rax
add rax,rdi
jmp .L0

.L0
[UNUSED]
sub rdi,rsi
retq
```

(g)

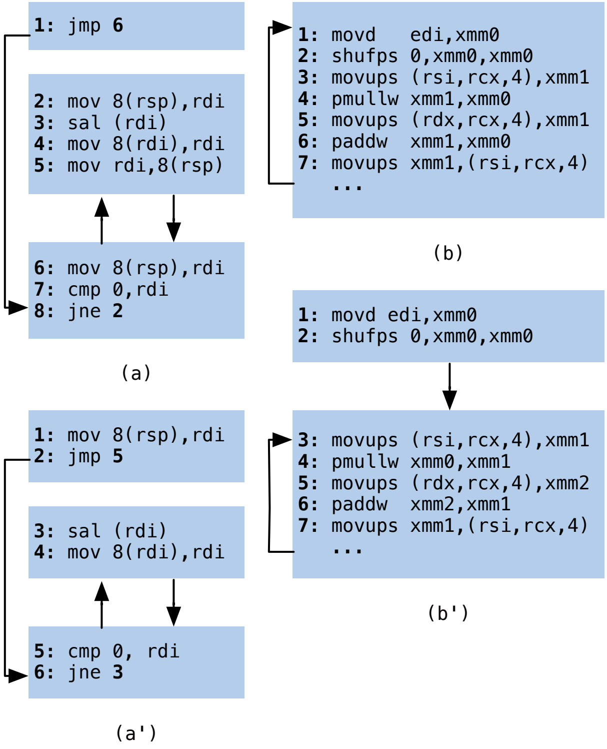
Figure 5. STOKE moves applied to a representative code (a). Instruction moves randomly produce the UNUSED token (b) or replace both opcode and operands (c). Opcode moves randomly change opcode (d). Operand moves randomly change operand (e). Swap moves interchange two instructions either within or across basic blocks (f). Resize moves randomly change the allocation of instructions to basic blocks (g).

sions are sufficient to allow for the exploration of rewrites which include loop-invariant code motion, strength reduction, and most other classic loop optimizations.

The cost function used by STOKE to evaluate transformations includes two terms, a correctness term and a performance term. We leave the correctness term described by [33] unmodified. Candidate rewrites are executed in a memory sandbox on a representative set of tests. Values contained in live memory and registers are compared against the corresponding values produced by the target and penalties are assessed for bits which are incorrect or appear in the wrong location. STOKE approximates the performance of a candidate rewrite by summing the expected latencies of its constituent instructions in nanoseconds. We modify this function to account for loop nesting depth, $\mathtt{nd}(\cdot)$ as follows, where w is a constant which we set to 20.

$$\mathtt{perf}(C) = \sum_{bb \in C} \sum_{i \in bb} E\big[\mathtt{latency}(i)\big] \cdot w^{\mathtt{nd}(bb)}$$

We now discuss the failure cases of STOKE: the benchmarks of [33] for which STOKE produces code inferior to `gcc -O3`. The other benchmarks of [33] are loop free and are not relevant here. Our modified version of STOKE is able to produce the two optimizations shown (simplified) in the lower half of Figure 6. The optimizations produced by the original version of STOKE are shown above for reference. Figure 6a is a linked-list traversal. The kernel iterates over the elements in the list until it discovers a null pointer and multiplies each element by two. Our modified version of STOKE is able to cache the value of the head pointer in a register across loop iterations. Figure 6b performs the BLAS vector function $a \cdot \vec{x} + \vec{y}$. Whereas the original version of STOKE is able to produce an optimization using vector intrinsics, our version is able to further improve on that optimization by first performing a register renaming and removing the invariant computation of a 128-bit constant from the loop. The code motion is not possible if the registers are not suitably renamed first.



```
1: jmp 6

2: mov 8(rsp),rdi
3: sal (rdi)
4: mov 8(rdi),rdi
5: mov rdi,8(rsp)

6: mov 8(rsp),rdi
7: cmp 0,rdi
8: jne 2
```

(a)

```
1: movd   edi,xmm0
2: shufps 0,xmm0,xmm0
3: movups (rsi,rcx,4),xmm1
4: pmullw xmm1,xmm0
5: movups (rdx,rcx,4),xmm1
6: paddw  xmm1,xmm0
7: movups xmm1,(rsi,rcx,4)
   ...
```

(b)

```
1: mov 8(rsp),rdi
2: jmp 5

3: sal (rdi)
4: mov 8(rdi),rdi

5: cmp 0, rdi
6: jne 3
```

(a')

```
1: movd edi,xmm0
2: shufps 0,xmm0,xmm0

3: movups (rsi,rcx,4),xmm1
4: pmullw xmm0,xmm1
5: movups (rdx,rcx,4),xmm2
6: paddw  xmm2,xmm1
7: movups xmm1,(rsi,rcx,4)
   ...
```

(b')

Figure 6. Simplified versions of the optimizations produced and verified by our modified version of STOKE. The iteration variable in (`a`) is cached in a register (`a'`). The computation of the 128-bit constant in (`b`) is removed from an inner loop (`b'`).

Performance data for these experiments are shown in Table 3. We note that the time spent verifying `bansal` is significantly greater than what was reported for the COMPCERT benchmarks. This is because STOKE produces 64-bit as opposed to 32-bit x86 binaries, which results in more complex memory equality constraints. The aim of our experiments is to describe the effectiveness of our core ideas and standard heuristics for constraint simplification, which we have not implemented, can be applied to achieve better performance [8]. To illustrate just how important constraint simplification is, for the SAXPY benchmark, following [42] we perform slicing on VCs to eliminate constraints that are not relevant to the verification task. The result is that verification requires less than a second, a significant improvement and the fastest verification time in our benchmark suite. Without these constraint simplifications, DDEC times out after four hours. This suggests that there is substantial room left for further performance optimization of DDEC's generated constraints; however such optimizations increase the size of the trusted code base, which is currently just the circuits for instructions, VC generator, and the theorem prover. Since STOKE starts optimizations from `llvm -O0` compilations, our verification results imply that STOKE+DDEC can produce binaries comparable to `gcc -O3` in performance and provably equivalent to unoptimized binaries generated by `llvm`.

6. Related Work

Our approach borrows a number of ideas from previous work on equivalence checking [6], translation validation [25], and software verification [34]. Combining these ideas with our technique for guessing the simulation relation from tests yields the first equivalence checking engine for loops written in x86.

Program equivalence checking is an old problem with references that date back to the 1950s. Equivalence checking is common practice in hardware verification, where it is well known that cutpoints play a critical role in determining equality. In sequential equivalence checking, for example, state-carrying hardware elements constitute cutpoints. Equivalence checking of low-level code has also been studied for embedded software [1, 6, 10, 11, 35]. None of these techniques support `while` loops and so are not applicable to our benchmarks.

Our goal is more ambitious than the state of the art in equivalence checking for general purpose languages: DDEC is a practically useful, automatic, and sound procedure for checking equivalence of loops. UC-KLEE [30] performs a bounded model checking that checks equivalence for all inputs up to a certain size. In another version of bounded model checking, differential symbolic execution [28] and SymDiff [20] bound the number of iterations of loops. Semantic Diff [17] checks only whether dependencies are preserved in two procedures. The approach in [23] handles neither while loops nor pointers. Regression verification [13] handles only partial equivalence: it does not deal with termination. As an alternative to DDEC, one can imagine composing two programs into a single program and then using an abstract interpreter [27]. However, the composition of [27] relies on syntactic heuristics that seem difficult to apply to binaries. Even if the composition step is successful, it is not clear how one would argue about the termination behaviors of the target and the rewrite (Definition 1) from the composed program and indeed [27] assumes terminating executions.

Fractal symbolic analysis [24] and translation validation [14, 25, 29] are two techniques that reason about loops in general. Both rely on information about the compiler, such as the specific transformations that the compiler can perform. Hence, these are not directly applicable to the problem of equivalence checking of code of unknown provenance. If one is simply given two assembly programs to check for equivalence there is no "translation" and hence translation validation is not applicable.

Conceptually, if one tries to port the approach of Necula [25] to x86, then one will need to build a static analysis for x86 which is sound and precise enough to generate simulation relations. This is a decidedly non-trivial engineering task and has never been done (see Section 4.6); DDEC side-steps this issue by using concrete executions (i.e., tests) for finding cutpoints and generalizing from tests to invariants. In addition, the constraints generated by symbolically executing x86 opcodes are complicated enough that we believe that the decision procedure of [25] will fail to infer equalities in most cases. Nonetheless, compiler annotations are a rich source of high level information, and as a result translation validation techniques can handle transformations that DDEC currently cannot, such as reordering traversals over matrices. For DDEC to handle such programs we would need better invariant inference algorithms that can infer quantified invariants fully automatically.

Generation of invariants from test data for verification was pioneered by Daikon [9]. In a previous work [34], we computed equality invariants for a single program using nullspace computations for a restricted language with assignments, branches, and loops. Similar to DDEC, the candidate invariants were checked by a theorem prover. We have shown that by choosing features appropriately, nullspaces can help infer simulation relations for the much more complex x86 language.

Some of the most recent work on equivalence checking includes random interpretations [15] and equality saturation [36, 38]. The former represents programs as polynomials which it requires to be of low degree. Unfortunately, bit-manipulations and other similar machine-level instructions are especially problematic for this technique. For example, a shift left by one bit has a polynomial of degree 2^{63} for the carry flag. Unlike equality saturation, DDEC does not rely on expert provided equality relationships between program constructs, which would be difficult to produce by hand for a CISC architecture such as x86. Further comparison with equality saturation is in Section 5.1.

Superoptimizations and the related synthesis task have previously been limited to sequences of loop-free code [4, 16, 18, 22]. Using DDEC, we have presented the first results for a superoptimizer able to synthesize provably correct loops.

7. Conclusion

In this paper we describe a data-driven procedure for verifying equivalence of two loops, using it to verify the equivalence of binaries produced by a certified compiler against those produced by an optimizing compiler and also extending a stochastic superoptimizer to perform loop optimizations. Other interesting applications are possible, such as checking that code refactoring preserves equivalence, and we hope to explore these in the future. The main limitations of the current implementation are restricted expressiveness of the inferred invariants and the inability to handle floating point instructions. With advances in invariant inference and decision procedures, we hope to remove both limitations.

Acknowledgments

We thank George Necula, Jan Vitek, and the anonymous reviewers for their constructive comments. This work was supported by the Army High Performance Computing Research Center and NSF grant CCF-0915766. This material is also based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck. Formal verification of backward compatibility of microcode. In *CAV*, pages 185–198, 2005.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[3] G. Balakrishnan and T. W. Reps. WYSINWYX: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.

[4] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, pages 394–403, 2006.

[5] Z. Chen and A. Storjohann. A BLAS based C library for exact linear algebra on integer matrices. In *ISSAC*, pages 92–99, 2005.

[6] D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic formal verification of DSP software. In *DAC*, pages 130–135, 2000.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[8] I. Dillig, T. Dillig, and A. Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *SAS*, pages 236–252, 2010.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[10] X. Feng and A. J. Hu. Automatic formal verification for scheduled VLIW code. In *LCTES-SCOPES*, pages 85–92, 2002.

[11] X. Feng and A. J. Hu. Cutpoints for formal equivalence verification of embedded software. In *EMSOFT*, pages 307–316, 2005.

[12] P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *PLDI*, pages 441–452, 2012.

[13] B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.

[14] B. Goldberg, L. D. Zuck, and C. W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electr. Notes Theor. Comput. Sci.*, 132(1):53–71, 2005.

[15] S. Gulwani. Program analysis using random interpretation. In *Ph.D. Dissertation, UC-Berkeley*, 2005.

[16] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[17] D. Jackson and D. A. Ladd. Semantic Diff: A tool for summarizing the effects of modifications. In *ICSM*, pages 243–252, 1994.

[18] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28(6):967–989, 2006.

[19] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. ISBN 0-201-03822-6.

[20] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *CAV*, pages 712–717, 2012.

[21] X. Leroy. The CompCert C verified compiler documentation and users manual, 2013. URL `http://compcert.inria.fr/man/manual.pdf`.

[22] H. Massalin. Superoptimizer - a look at the smallest program. In *ASPLOS*, pages 122–126, 1987.

[23] T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In *ISQED*, pages 370–375, 2006.

[24] V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):776–813, 2003.

[25] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, 2000.

[26] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, pages 683–693, 2012.

[27] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *SAS*, pages 238–258, 2013.

[28] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.

[29] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, pages 151–166, 1998.

[30] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, pages 669–685, 2011.

[31] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.

[32] M. Rinard. Credible compilers. Technical report, Massachusetts Institute of Technology, 1999.

[33] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, pages 305–316, 2013.

[34] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592, 2013.

[35] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *CC*, pages 221–236, 2005.

[36] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL*, pages 264–276, 2009.

[37] A. V. Thakur and T. W. Reps. A method for symbolic computation of abstract operations. In *CAV*, pages 174–192, 2012.

[38] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, pages 295–305, 2011.

[39] A. Turing. Checking a large routine. In *The early British computer conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

[40] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201914654.

[41] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.

[42] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363, 2005.