Synthesizing Loops For Program Inversion

Cong Hou[1], Daniel Quinlan[2], David Jefferson[2],
Richard Fujimoto[1], and Richard Vuduc[1]

[1]Georgia Institute of Technology
[2]Lawrence Livermore National Laboratory
hou_cong@gatech.edu,dquinlan@llnl.gov,jefferson6@llnl.gov
fujimoto@cc.gatech.edu,richie@cc.gatech.edu

Abstract. We propose a new automatic program inversion method for imperative programs that contain loops. In particular, given a loop that produces an output state given a particular input state, our method can synthesize an inverse loop that reconstructs the input state given the original loop's output state. The synthesis process consists of two major components: (a) building the inverse loop's body, and (b) building the inverse loop's predicate. Our method works for all natural loops, including those that take early exits (e.g., via breaks, gotos, returns). This work extends a program analysis and synthesis framework, called Backstroke[1], that we developed in prior work.

Keywords: Program Inversion, Program Synthesis, Compilers

1 Introduction

We consider the problem of synthesizing *program inverses*. That is, given a program P with input state I and output state O, its *inverse* or *reverse program*, P^-, produces I given O. Our primary motivation comes from optimistic parallel discrete event simulation (OPDES). There, a simulator must process events while respecting logical temporal event-ordering constraints; to extract parallelism, an OPDES simulator may speculatively execute events and only *rollback* execution when event-ordering violations occur [4]. In this context, the ability to perform rollback by running time- and space-efficient P and P^-, rather than saving and restoring large amounts of state, can make OPDES more practical. Synthesizing inverses also appears in numerous other software engineering contexts, such as debugging [1], synthesizing "undo" code, or even generating decompressors automatically given only lossless compression code [10]. The challenge in any of these contexts is that constructing program inverses manually is a tedious, time-consuming, and error-prone task.

The program P will generally contain non-invertible statements, such as a destructive assignment. However, in these cases it may still be possible to create an inverse. In particular, we may create an instrumented version of P, called

[1] As a sub-project of the ROSE compiler Backstroke can be downloaded here: http://www.rosecompiler.org.

the *forward program*, P^+, with semantics-preserving changes so that it becomes possible to construct P^- from P^+. We then replace executions of P with P^+. For instance, suppose P overwrites a variable v. We may construct P^+ so that it saves the value of v prior to overwriting it. Then, P^- need only restore the saved value to recover v's prior value. In this case, P^+ produces extra outputs, which we denote by S. Indeed, even if P is theoretically reversible without requiring extra output, we may nevertheless need to generate S due to fundamental technical limits on program analysis.

This paper extends our prior compiler-based program inversion framework, called Backstroke [3], to handle the case of programs with loops. As a concrete introductory example, consider the following program on the left, which takes as input an integer $n \geq 0$ and produces the output, $s \leftarrow \sum_{i=0}^{n} i$:

```
s = 0;                          n = 0;
while (n > 0) {                 while (s > 0) {
    s = s + n;                      n = n + 1;
    n = n - 1;                      s = s - n;
}                               }
```

Our goal is to construct an inverse P^- that recomputes n given s, as shown above on the right. Our initial work on Backstroke proposed two new intermediate program representations, which we call the *value search graph* and *route graph* representations. However, these representations could not represent loop structure and hence could not generate the inverse shown above. The method of this paper can; additionally, it can recognize certain special cases where synthesis of an explicit forward program P^+ can be avoided, as is the case in this example. Such special cases are often a requirement in general software engineering (as opposed to OPDES) contexts.

The above example is special in that the loop has single-entry and single-exit points. Consequently, in the usual control-flow graph (CFG) analysis used inside a compiler, the loop's inputs and outputs are easy to identify and program analysis becomes simpler, because the compiler may analyze the loop in relative isolation from the rest of the program. To handle more complex loops, such as those with multiple exits via `break` or `return`, we show how we can modify the CFG to reduce it to the preceding simpler form (Section 3.2). Thus, our method readily applies to the class of so-called *natural* loops.

Note that we use the terms, "program inverse" and "program inversion," even though strictly speaking an inverse for P (as opposed to the instrumented forward program, P^+) may not exist. Nevertheless, this terminology is standard in our OPDES context, so we adhere to it in this paper [11].

2 Prior Foundations: Value Search and Route Graphs

Our work on program inversion for loops builds on a program analysis and synthesis framework that we developed in our prior work. As noted previously, the framework comprises two novel intermediate program representations, which we refer to as *value search graph* and *route graph* [3]. This section summarizes the key ideas behind these representations, explaining how we use them to construct both forward and reverse programs. (Please see our earlier paper for all the formal details [3].) Section 3 describes our extensions for loops.

The basic program inversion workflow in our framework is as follows. Given P, we first translate the program into a standard compiler intermediate program representation known as single static assignment (SSA) form [2]. From the SSA form, we construct a value search graph (VSG) [3]. The problem of finding a forward or reverse program becomes a combinatorial search problem on the VSG. The result of this search is a subgraph of the VSG, which we call a route graph (RG) [3]. There may be many such search results, each of which is a particular forward or reverse program. Lastly, from the RG we synthesize the actual code that implements the forward or reverse program. The process is illustrated in Figure 1. We elaborate on the process and discuss the example next.

The VSG essentially expresses *equality* relations between values in the program. Given these relations, we can determine how values from the input I eventually relate to the values produced during the execution of the forward program P^+, such as the values in the output O. To get the relations, we first transform the program into SSA form. The SSA form is semantically equivalent to the original program but has the special property that each variable is defined only once. In the VSG, nodes represent constants, variables from the SSA, and operators (e.g., "+" or "−" operations); directed edges represent either equality or operand-operator relationships. Edges are also annotated with information about the control-flow paths on which the particular equality relation exists, allowing us to handle conditional branches. Lastly, if there is no way to retrieve a desired value from computational operations alone, we will need to save that value during the execution of P^+ so we can later retrieve it in P^-. Such a *state saving* operation becomes an additional type of node in the VSG. Since state saving may incur both time and space overheads to P^+, we can add a suitable cost to each edge incident to the state saving node.

Given the VSG, we locate *target nodes*, which contain all values we wish to compute. For instance, if we want to build P^- and reconstruct a particular value from the original input I, the corresponding node for that value in the VSG becomes a target node. During the analysis of the VSG, some nodes will be considered *available*. For example, when building P^-, nodes containing constants, the state saving node, and final outputs O of the original program are available. Starting from the targets, we perform a path search through the VSG looking for available nodes. The result of this search is a subgraph of the VSG, which we call a *route graph* (RG). The search algorithm works in such a way that it guarantees each value is retrieved only once for each control flow graph (CFG) path. (The search for a RG that minimizes state-saving cost is NP-Complete, which our prior paper both proves and provides heuristics to find low-cost solutions [3].)

Finally, we generate P^+ and P^- from the RG. In the RG, for each edge pointing to the state saving node, we will instrument P with a state saving statement storing the corresponding value. Also, we use a bit vector to record the control flow paths in P. Then P^+ is generated by instrumenting P with statements performing state savings and CFG path recording. To generate the reverse program P^-, we build a CFG for the reverse function from the RG, and P^- is generated from the CFG.

```
int a, b;
void foo() {
  if (a == 0)
    a = 1;
  else {
    b = a + 10;
    a = 0;
  }
}
```

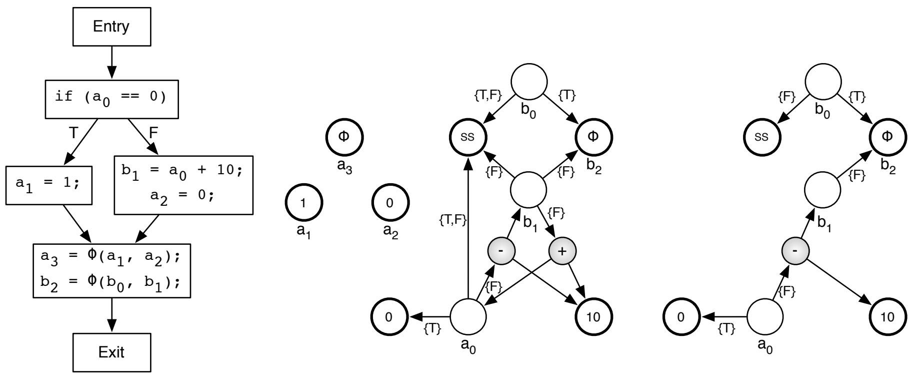
(a)

```
void foo_forward() {
  int trace = 0;
  if (a == 0) {
    trace |= 1;
    a = 1;
  }
  else {
    store(b);
    b = a + 10;
    a = 0;
  }
  store(trace);
}
```

(b)

```
void foo_reverse() {
  int trace;
  restore(trace);
  if ((trace & 1) == 1)
    a = 0;
  else {
    a = b - 10;
    restore(b);
  }
}
```

(c)



(d) (e) (f)

Fig. 1. (a) The original program. (b) The forward program. (c) The reverse program. (d) The CFG in SSA form. (e) The VSG. Nodes in bold are available nodes and all outgoing edges are removed from them. (f) The RG.

Figure 1 illustrates the entire process in a loop-free example. The original program is the function `foo`. The variables a and b are both inputs and outputs. The CFG in SSA form is shown in Figure 1(d). In SSA form, the input of this program are a_0 and b_0, and the output are a_3 and b_2; note that the original variables have subscripts in SSA form, which are referred to as *versions* of the original variables. Observe that versioned variables are in the static program assigned only once. (Programs with loops will need special treatment and extension.) From the SSA CFG, we then build the VSG shown in Figure 1(e). The "SS" node is a special state-saving node. All outgoing edges from each available node shown in bold are removed since the search always ends at available nodes.

Each equality relation (edge) is constrained by a set of CFG paths. Since there are only two paths in the program, we use T and F to represent the path passing through the true and false bodies, respectively. Our goal is to retrieve a_0 and b_0, which is done by searching the VSG to find a way to get its value for each CFG path. The search result, which is the RG, appears in Figure 1(f). We build the forward and reverse programs, Figures 1(b) and (c), respectively, from the RG. (For details on this process, refer to our prior paper [3].)

3 Handling loops

Unmodified, our prior method as described in Section 2 cannot handle loops for two key reasons. First, a loop results in cyclic paths in the CFG, whereas our prior analysis relies on paths being acyclic. Acyclic paths make it easy to check that the reverse program restores any desired input value no matter what path the forward program takes. Secondly, our prior VSG and RG cannot represent loop control structure. Therefore, it is simply not possible to synthesize, for example, a loop in the reverse code from the RG. Nevertheless, we *can* reuse most of the prior method by decomposing the problem suitably. In particular, we keep the basic framework of "SSA to VSG to RG." Our extension replaces SSA with a loop-enabled variant, and then extends our VSG and RG representations and algorithms to deal with cycles, thereby addressing the two aforementioned issues.

Let us first assume that each loop to be reversed is a single-entry, single-exit while loop (we will explain what is a while loop later). We explain in Section 3.2 how to convert other kinds of loops into this form. We also assume that each loop must terminates at run-time so that we can always get an output. Given an input while loop, there are three steps to build a VSG.

1. We temporarily collapse each while loop into a single abstract node in the CFG, thereby creating a logically loop-free CFG from which we can build a VSG by directly applying our prior method. This "transformation" is for program analysis purposes only. We denote this loop-collapsed VSG by G_P.
2. Similarly, we directly apply our prior method to build a VSG for each loop *body*, which may be treated as another loop-free program. (If the body contains nested loops, these are similarly collapsed as in Step 1 above.) Note that path information in these loop body VSGs are local to the loop body. We denote this VSG for the loop body by G_L.
3. At this point, G_P and G_L are disconnected. Therefore, we introduce new special edges to connect them, thereby resulting in a single connected VSG. These connecting edges are a new type of edge and constitute the main extension to our prior VSG in order to support loops. The new edges connect each input (or output) of a loop to the input (or output) of the loop's body. These new edges serve as markers: when we search the VSG and produce an RG containing these edges, then we know we need to synthesize a loop.

Since Steps 1 and 2 use our prior VSG construction, we need not discuss them further here. What changes is the third step, as detailed below, including new

VSG searching rules and new procedures for synthesizing loops from the search result (i.e., the RG).

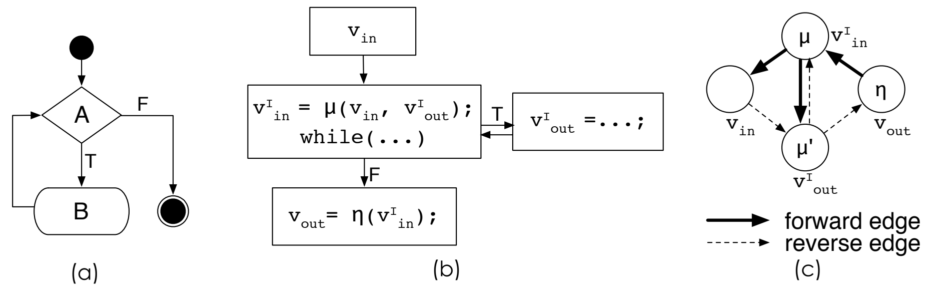


Fig. 2. (a) The diagram of a while loop. (b) The CFG in loop-closed SSA form for a variable v modified in the loop. (c) Forward and reverse edges.

3.1 Dealing with while loops

We first consider a while loop with the diagram shown in Figure 2(a). We further assume that A has no side-effects and that there are no escapes from B. Thus, the loop only exits from its entry.

Given such a while loop, we transform it into the *loop-closed SSA form* [9], illustrated in Figure 2(b). Loop-closed SSA differs from conventional loop-free SSA as follows. In conventional SSA, a special marker called a ϕ *function* is placed in the CFG at the first program point where two distinct versions (definitions) of a variable, computed along different program paths, meet. In loop-closed SSA, if a value is defined inside of a loop and used outside of it, we place a special single entry ϕ function at the *exit* of the loop. To distinguish this type of loop-specific ϕ function from a conventional ϕ function as used in loop-free programs, we denote the loop-specific form by the term η function, by convention [7]. Additionally, suppose a definition of a variable from outside the loop and a definition coming from a back-edge of the loop meet at a program point. Again, we create a ϕ function marker here, and to distinguish it, we refer to it as a μ function.

To see how these markers work, consider a variable v modified by a while loop; we now describe the corresponding loop-closed SSA form, which Figure 2(b) illustrates. Let v_{in} denote the input value of v before the loop executes, and v_{out} the output value of v after the loop executes. Next, let the input to the loop *body* be v^I_{in} and the output v^I_{out}. (The superscript I is intended to remind the reader that these are values associated with an *iteration* of the loop, as opposed to the values before and after the loop.) Then, v^I_{in} is defined by a μ function as $v^I_{\text{in}} = \mu(v_{in}, v^I_{out})$, and v_{out} is defined by a η function as $v_{\text{out}} = \eta(v^I_{in})$. That is, $v^I_{\text{in}} = \mu(v_{in}, v^I_{out})$ indicates the program point at which v has either the initial value before the loop executes or the value produced by some iteration of the loop; and $v_{\text{out}} = \eta(v_{in}^I)$ indicates the program point at which v has the final value once the loop completes.

From this loop-closed SSA form, we wish to build a VSG that will express equality relations among the four SSA values, v_{in}, v_{out}, v_{in}^I, and v_{out}^I. This VSG result is shown in Figure 2(c). Recall that nodes in the VSG represent values, and edges the equality relations. There are four value nodes. The nodes v_{in} and v_{out} are part of the loop-collapsed G_P, and v_{in}^I and v_{out}^I belong to the loop body's G_L. The μ and η functions indicate how to connect G_P and G_L. In particular, the three solid bold edges are associated with the dependences induced by executing the loop in the forward direction; we call these the *forward edges*, and a μ node is incident to all three. The presence of these edges make it possible to obtain v_{out} by some path passing through G_L, and simultaneously indicate that a loop is present for subsequent code generation. Similarly, the three dashed edges are *reverse edges* associated with dependences induced in the reverse direction. These edges make it possible to obtain v_{in} by some path through G_L. Note that the reverse edges form a symmetry to the forward edges. From this symmetry, we define the node incident to all three reverse edges as a μ' node. Later we will show how the search traverses these edges.

Having built the CFG, the next step is to search it, producing the RG result. Recall from Section 2 that we are given a set of target nodes whose values we wish to eventually compute from a starting set of available nodes. We search for a path from available nodes to target nodes; the subgraph representing paths is the RG, which is not necessarily unique. Our algorithm is similar to the one we have described previously [3], but for loops we need three additional search rules:

– During a search for a value, once a forward/reverse edge is selected, all edges in the other category cannot be chosen. This is because either a forward or a reverse loop will be built to retrieve the value.
– When the search reaches a μ or μ' node, it will be split into two sub-searches, in G_P and G_L, respectively, through the two outgoing forward or reverse edges. For example, in Figure 2(c), if the search reaches v_{in}^I, the algorithm begins two sub-searches beginning with v_{in} and v_{out}^I.
– During the search, the algorithm may form a directed cycle only in G_L; furthermore, such a cycle must contain a forward or reverse edge between a μ and μ' node. Once a cycle is formed, the search in G_L is complete.

We build a while loop as either a forward or a reverse loop. Synthesizing such a while loop consists of synthesizing its body and predicate.

Building the loop body. The loop body in the reverse program is generated from the search result in G_L. For each variable we remove the edge between the μ and μ' nodes and hence remove the cycles, so that we can generate the loop body using our prior code generation algorithm [3].

Building the loop predicate. To guarantee that the generated loop has the same iterations at runtime as the original loop, we need to build a proper loop predicate. We propose three approaches to building a correct loop predicate. To illustrate those approaches, we temporarily introduce the following loop example. We assume that the omitted statements modify neither `A[]` nor `i`.

```
i = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
}
```

- **Approach 1:** Building the same loop predicate as that in the original loop. To build this predicate, we need to retrieve each value in the predicate. A new search is needed to acquire those values, and the search result will be combined into the RG generated above. For the example above, we can build a loop below that has the same number of iterations as the original one. The omitted statements will be substituted by the loop body built above.

```
i = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
}
```

- **Approach 2:** Building the loop predicate from a variable updated in the loop. Given a variable v and its four definitions: v_{in}, v^I_{in}, v^I_{out}, and v_{out}, if $v^I_{\text{in}} \neq v_{\text{out}}$ in each iteration except the last definition of v^I_{in} (which is actually v_{out}), and if we can retrieve v_{in} and v_{out} before the loop (hence we cannot retrieve them through the loop), and v^I_{out} in the loop, we can use them to build a while loop as:

$$u := v_{in};\ while(u \neq v_{out})\ \{\ /*\ update\ u\ */\ \}$$

Similarly, if $v^I_{\text{out}} \neq v_{\text{in}}$ in each iteration, and v_{in} and v_{out} can be retrieved before the loop, and v^I_{in} can be retrieved in the loop, we can use them to build a while loop as:

$$u := v_{out};\ while(u \neq v_{in})\ \{\ /*\ update\ u\ */\ \}$$

In general, it is difficult to detect all variables satisfying the properties above. However, there are some special cases. One case is that of *monotonic variables* [12], which are monotonically strictly increasing or decreasing in each iteration. Another is that of induction variables, which are special monotonic variables that are relatively easier to recognize. In the above example, `i` is an induction variable. Assume its final value after the loop is `i1` that is known, and then we can build the following loop with the predicate using `i`.

```
i = 0;
while (i != i1) {
    /* ... */
    i = i + 2;
}
```

- **Approach 3:** Instrumenting the original loop with a counter counting the number of iterations. The counter has the initial value zero and is incremented by one on each back edge of the loop. The final value of the counter is stored in the forward program and restored in the reverse program as the maximum value of another loop counter. This approach generally works if either of the above two approaches fail. However, it requires instrumentation (the counter), and therefore forces generation of a forward program. Below we show the instrumented loop in the forward program (left) and the generated loop in the reverse program (right) for the above example.

```
i = count = 0;                  restore(count);
while (A[i] > 0) {              while (count > 0) {
    /* ... */                       /* ... */
    i = i + 2;                      count = count - 1;
    count = count + 1;          }
}
store(count);
```

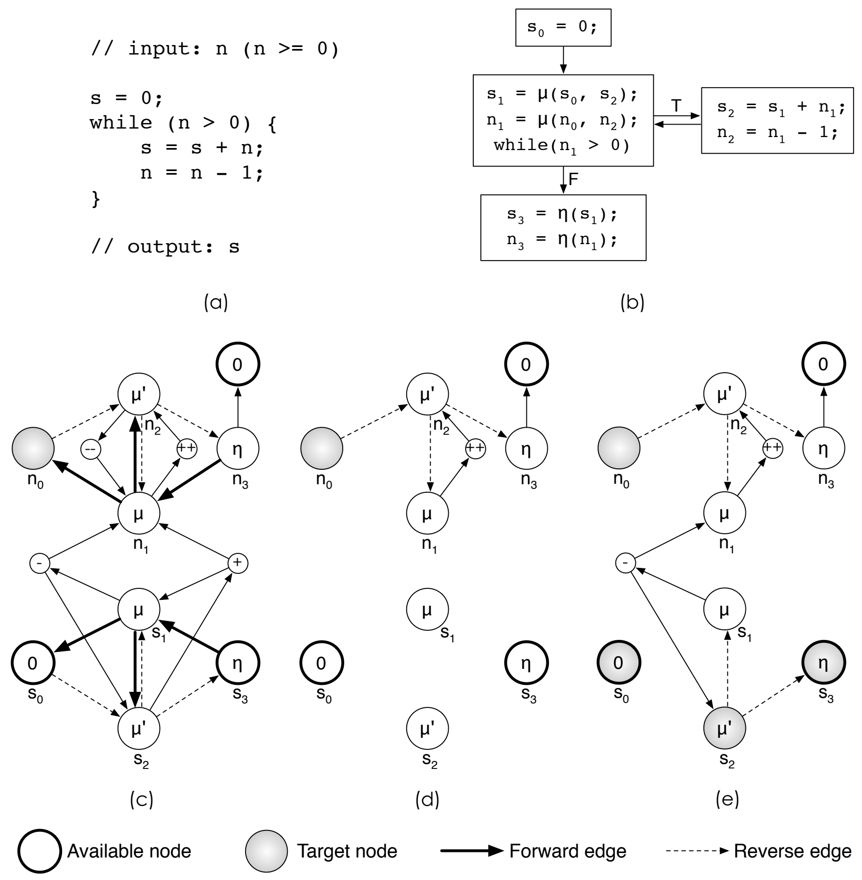
We prioritize these approaches as follows. Applicability and state-saving cost are our main criteria. We prefer Approach 1 and 2 over 3. When either 1 or 2 apply, if no state-saving is required, we apply them. Otherwise, we try Approach 3 and choose the overall approach with the least cost.

As an example, suppose we apply this algorithm to the loop in Figure 3(a). Figure 3(b) shows its CFG in loop-closed SSA. The input is n_0 and the output s_3. Our goal is to generate a reverse program that takes s_3 as input and produces n_0. We build the VSG shown in Figure 3(c), with forward and reverse edges shown as bold and dashed edges, respectively. Note that the equality between n_3 and 0 is acquired from solving constraints, a standard compiler technique, as discussed in Section 3.3.[2] The search result for value n_0 is shown in Figure 3(d), from which we can build the loop body as { `n = n + 1;` }.

Next, we build the loop predicate. In our example, because we wish to retrieve the initial value of n, we cannot use it to build the loop predicate. We can discover that s is a monotonic variable, and that both the initial and final values of s, which are 0 and s_3, respectively, are available. To get s_2, we search its value on the VSG and the search result is shown in Figure 3(e). As a result, we build the loop predicate from s and the reverse program is generated as below.

```
n = 0;
```

[2] For clarify, we remove the equality $n_1 = s_2 - s_1$, as this relation will not be used during the search.



```
// input: n (n >= 0)

s = 0;
while (n > 0) {
    s = s + n;
    n = n - 1;
}

// output: s
```

Fig. 3. (a) The program of our example. (b) The CFG in loop-closed SSA form. (c) The VSG. (d) The RG for retrieving n_3. (e) The RG for retrieving n_0 and s_2.

```
while (s != 0) {
    n = n + 1;
    s = s - n;
}
```

3.2 Dealing with loops other than while loops

In practice, the vast majority of loops have a single entry, which are called *natural loops* [5]. Loops with more than one entry are quite rare and can in fact be transformed into natural loops [5]. However, it is quite common that a loop has several exits. For example, in C/C++ we may exit a loop early through `break`, `return`, or `goto` statements. Nevertheless, given a non-while natural loop, we can transform it to separate the last iteration from the loop;

then, the remaining iterations form a new while loop, and the last iteration will not belong to the loop and hence can considered with the control flows outside of the loop. We then process the new while loop as previously described. Note that this "transformation" is only applied to the CFG during the analysis, and not to the original program. As such, in the forward program P^+ the last iteration and other iterations of each loop continue to share the same code.

Figure 4(a) shows a loop in a CFG, with a header (node 1) and two back edges ($4 \to 1$ and $5 \to 1$). There are two different exits from this loop, which are nodes 6 and 7. Figure 4(b) shows the CFG of the transformed loop. This transformation is performed as follows.

In a natural loop, only the last iteration takes the exit, and any other iteration goes back to the loop header. Therefore, if the last iteration is peeled off from the loop, this loop will turn into a while loop. To implement this transformation, we create a new branch node with an unknown predicate that returns $true$ if the next iteration is not the last one and $false$ otherwise. Note that we will not build this predicate in the forward program. The new branch node turns over all in-edges of the loop header. Its $true$ labeled out-edge will point to the loop header of a copy of the loop (node $1'$) with back edges but without exit edges, and all back edges are redirected to this new branch node making it a new loop header. Note that after removing exit edges it is possible that a previous branch node becomes a non-branch node (node $3'$, for example), which is fine because the removed branch edge will not be taken. Then, we can remove the (side effect free) predicates from those nodes. The edge labeled with $false$ from the new branch node will point to the original loop header (node 1) and all back edges in the original loop are removed, since the last iteration won't take the back edge. The nodes from which the exit of the program is not reachable due to the back edge removal are removed (node 4 and 5, for example). Again the predicate is removed from a node once it is not a branch node anymore (node 2 and 3).

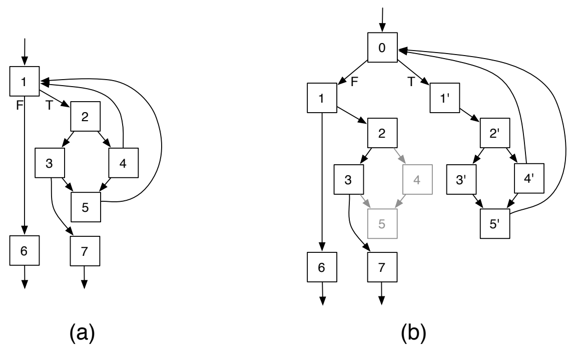


Fig. 4. (a) A loop in CFG with two back edges and two exits. (b) The CFG of the transformed loop.

After the transformation, all loops in the program become while loops and our method applies.

3.3 Discussion

Equality from solving constraints. We use constraint solving to obtain any needed equalities. For example, if $a \geq b$ and $a \leq b$, then $a = b$. This method is useful to get the final value of a loop counter. A typical example is shown below:

```
i = 0; while (i < N) { ...; i = i + 1; }
```

where i is a loop counter incremented by one in each iteration, and $N \geq 0$. In Floyd-Hoare logic [6], the partial correctness of a while loop is governed by the following rule of inference [8]:

$$\frac{\{C \wedge I\} \text{ body } \{I\}}{\{I\} \textbf{ while } (C) \text{ body } \{\neg C \wedge I\}}$$

where C is the while condition, and I is a loop invariant, which is informally defined as a statement of the conditions that should be true on entry into a loop and are guaranteed to remain true on every iteration of the loop. In this example, we choose $i \leq N$ as a loop invariant. After replacing C and I with $i < N$ and $i \leq N$, the postcondition at the end of the loop $\{\neg C \wedge I\}$ becomes $\neg(i < N) \wedge i \leq N$, from which we get $i = N$.

Rebuilding control flows for the reverse program. In our prior work [3], we record the runtime control flow paths in the forward program using a bit vector. Specifically, a bit is used to record which path is taken at each two-way branch node. The bit vector is stored at the end of the forward program and is used to rebuild the control flows in the reverse program. This method has both low time and space overhead. However, for program with loops, recording control flows in each iteration of a loop may bring considerable space overhead.

To avoid this overhead, we found that we could calculate the control flows instead of storing and restoring them. Basically, there are two ways to do that. First, for a predicate in the original program, we can recover all values used in the predicate in the reverse program, then use those values to produce the result of the predicate. Second, if there is a ϕ function defined at a join node in the original program as $v_2 = \phi(v_0, v_1)$, and v_0 and v_1 cannot have the same value, then if we can get the value range of v_0 or v_1 and retrieve the value of v_2 , we can build a predicate by checking if the value of v_2 is in the value range of v_0 or v_1. More details will be introduced in our future publications.

4 Conclusion and future work

With our loop handling methods, Backstroke can now handle a variety of programs that operate on scalar variables. The next major direction for this work are to handle array programs and programs that manipulate complex data structures, such as linked data structures. However, our work to date lays the critical foundations for such extensions; for example, the index of an array is a scalar that can be retrieved in a loop using the method described in this paper. Our future work will focus on synthesizing reverse programs that use arrays and object accesses. We are particularly interested in whether our techniques can be used to reverse programs known to be reversible by computation, such as lossless compression and decompression, encryption and decryption, among numerous others.

Acknowledgements

This work was supported in part by the National Science Foundation (NSF), under CAREER award number 0953100, and grants from the U.S. Department of Energy (DOE) through Lawrence Livermore National Laboratory (LLNL) LDRD project 10-ERD-025. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, DOE, or LLNL.

References

1. B. Biswas and R. Mall. Reverse execution of programs. *ACM SIGPLAN Notices*, 34(4):61–69, Apr. 1999.
2. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
3. C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc. A new method for program inversion. *Compiler Construction*, 2012.
4. D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
5. S. S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.
6. V. R. Pratt. Semantical consideration on floyo-hoare logic. *17th Annual Symposium on Foundations of Computer Science*, 1976.
7. Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Language. In *PLDI 1990*, 1990.
8. G. Rosu, C. Ellison, and W. Schulte. Matching logic: An alternative to hoare/floyd logic. *Proceedings of the 13th international conference on Algebraic methodology and software technology*, 2010.
9. Sebastian Pop, Pierre Jouvelot, and Georges-Andre Silber. In and Out of SSA : A Denotational Specification. *Static Single-Assignment Form Seminar*, 2009.
10. S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI '11*. ACM Press, 2011.
11. G. Vulov, C. Hou, R. Vuduc, D. Quinlan, R. Fujimoto, and D. Jefferson. The backstroke framework for source level reverse computation applied to parallel discrete event simulation. *Winter Simulation Conference*, 2011.
12. M. Wolfe. Beyond Induction Variables. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation (PLDI)*, 1992.