A New Method for Program Inversion

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

[1] Georgia Institute of Technology, Atlanta, GA 30332
[2] Lawrence Livermore National Laboratory, Livermore, CA 94551
{hou_cong,georgevulov}@gatech.edu
{dquinlan,jefferson6}@llnl.gov
{fujimoto,richie}@cc.gatech.edu

Abstract. Program inversion has been successfully applied to several areas such as optimistic parallel discrete event simulation (OPDES) and reverse debugging. This paper introduces a new program inversion algorithm for imperative languages, and focuses on handling arbitrary control flows and basic operations. By building a value search graph that represents recoverability relationships between variable values, we turn the problem of recovering previous values into a graph search one. Forward and reverse code is generated according to the search results. We have implemented our algorithm as part of a compiler framework named Backstroke, a C++ source-to-source translator based on ROSE compiler. Backstroke targets optimistic simulation codes and automatically generates a reverse function to recover values modified by a target function. Experimental results show that our method is effective and produces better performance than previously proposed methods.

Keywords: Program inversion, SSA, SSA graph, reverse computation, state saving, ROSE.

1 Introduction

We consider the problem of how to generate an efficient *inverse* of a program. Informally, suppose a program P begins in state I and ends in state F. Then, an inverse, P^{-1}, reproduces the initial state I when started in state F. State I may not be uniquely reproducible given state F, hence P may have to be instrumented so that the original state I can be restored. The challenge in program inversion is how to instrument P and construct P^{-1} so that, when executed, they incur minimal storage and time overheads. This paper describes novel program analysis and code generation techniques for automatically building the instrumented P and a space- and time-efficient P^{-1}.

Program inversion has numerous applications, but our focus is on *optimistic* parallel discrete event simulations (OPDES). In this context, parallelization is achieved by speculatively executing each event in parallel and using rollback mechanisms to undo any events that have executed out of order [16,14].[1] In

[1] Every event has a *timestamp*, which establishes a total order on events [14].

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 81–100, 2012.

a naïve implementation of OPDES, we might instrument P to save the initial values of all variables that it modifies, so that P^{-1} simply restores those initial values. However, for events that manipulate a significant amount of state, this approach could incur significant overheads in both time and storage.

The alternative approach that we consider is *reverse computation* [8]. The idea is to minimally instrument P to store just enough control and data information so that, on rollback, a P^{-1} can computationally reconstruct the initial state through algebraic manipulations. Since on modern systems simple algebraic operations are much cheaper than memory accesses, reverse computation can be much more efficient than naïvely saving state. Figure 1 shows an example of an event, and its instrumented forward and reverse (inverse) versions (`a` and `b` are two state variables); the forward and reverse versions were generated by our algorithm. Rather than save both state variables, we can through reverse computation store just `b` in one branch.

```
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)

Fig. 1. (a) The original event (b) The forward event (c) The reverse event

Contributions. We build a graph that shows explicit relationships between values and allows us to restore values by searching the graph; costs on the edges of this graph correspond to memory overheads in the generated code. This approach allows us to flexibly mix state saving and reverse execution depending on which is more efficient; other approaches focus on either one or the other. Although the search problem is NP-complete, we provide heuristics that work well in practice. We have implemented our approach in the reverse compiler Backstroke, which automatically generates forward and reverse code as in Figure 1.

Limitations. In this paper we only target programs with scalar data types and without function calls. We do not address aliasing, arrays, and structured types; however we think the graph search approach to program inversion can be extended to these scenarios and provide similar benefits to the scalar case. While our cost model accounts only for memory costs, our approach is transparent to the cost model chosen; in the future, a more sophisticated cost model can be built to include other kinds of overhead.

2 Related Work

Most of the work on inverting arbitrary (non-injective) imperative programs has focused an incremental approach: the imperative program is essentially executed in reverse, with each modifying operation in the original execution being undone individually. For example, if statements $s_1 s_2 \dots s_n$ are executed in the forward directions, the reverse function executes statements $s_n^{-1} \dots s_2^{-1} s_1^{-1}$. The incremental approach cannot handle unstructured control flows and is difficult to apply with early returns from functions; the approach presented in this paper suffers from neither of these shortcomings. Furthermore, the incremental inversion restores the initial state by restoring every intermediate program state between the final state and the initial state, even though these states are not needed.

Among the incremental inversion approaches, syntax-directed approaches apply only statement-level analysis. If an assignment statement is lossless, its inverse is used: for example, the inverse of an integer increment is an integer decrement. Otherwise, the variable modified in the assignment has to be saved. An early example of syntax-directed incremental inversion is Brigg's Pascal inverter [7]. This approach was later extended to C and applied both to optimistic discrete event simulation [8] and reversible debugging [6].

Akgul and Mooney introduced a more sophisticated incremental inversion algorithm that uses def-use analysis to invert some assignment statements that are not lossless [2]; we refer to this approach as regenerative incremental inversion. In order to reverse a lossy assignment to the variable a, such as $a \leftarrow 0$, the regenerative algorithm looks for ways to recompute the previous value of a. One technique to obtain the previous value of a is to re-execute its definition; another technique is to examine all the uses of a and see if its value can be retrieved from any of its uses. These two techniques are applied recursively whenever a modifying operation is to be reversed; if they fail to produce a result, the overwritten variable is saved during forward execution. Our approach takes advantage of all the def-use relationships utilized by regenerative inversion, without suffering from the drawbacks of incremental inversion. In addition to def-use information, our approach also derives equality relationships between variables from the outcome of branching statements that test for equality or inequality.

A related line of work is inverting programs that are injective, without using any state saving. Most such work focuses on inverting functional programs [1,15,17]. Approaches to inverting imperative programs include translation to a logic language [20] and template-based generation of inverse candidates using symbolic execution [21].

3 Problem Setup

Let the set of *target variables* be $S = \{s_1 \dots s_n\}$ with *initial values* $V = \{v_1 \dots v_n\}$, where v_i is the initial value of s_i. These variables are modified by a *target function*[2] M, producing $V' = \{v'_1 \dots v'_n\}$, the *final values* of the target

[2] The function here is a C/C++ function, not a function in mathematics.

variables. Our goal is generating two new functions, the *forward function* M^S_{fwd} and the *reverse function* M^S_{rvs}, so that M^S_{fwd} transfers V to V', and M^S_{rvs} transfers V' to V. We define *available values* as values which are ready to use at the beginning of M^S_{rvs}. For example, values in V' and constants are available values. We also call values in V *target values* which are values we want to restore from M^S_{rvs}.

Note that M and M^S_{fwd} have the same input and output, but M^S_{fwd} is instrumented to store control flow information and values that are later used in M^S_{rvs}. This introduces two kinds of cost that must be considered when generating the forward-reverse pair $\{M^S_{fwd}, M^S_{rvs}\}$: extra memory usage and run-time overhead.

4 Reversing Functions without Loops

4.1 Framework Overview

We will first treat the inversion of loop-free code with only scalar data types, without aliasing. When such code is converted to static single assignment (SSA) form [11], each versioned variable is only defined once and thus there is a one-to-one correspondence between each SSA variable and a single value that it holds. We will also take advantage of the fact that loop-free code has a finite number of paths. Loops will be discussed in the next section, and non-scalar data types and aliasing will not be handled in this paper.

Given a cost measurement, for each path in the target function there should exist a best strategy to restore target values. Strategies usually vary among different paths. Therefore, the reversed function we produce should include the best strategy for each path; each path in the original function should have a corresponding path in the reverse function.

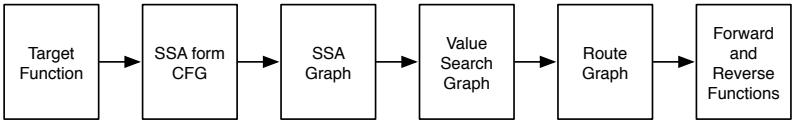


Fig. 2. Overall framework of the inversion algorithm

To restore target values, we will build a graph which shows equality relationships between values. We call this graph the *value search graph*, and it is built based on an SSA graph [3,10]. Then a search is performed on the value search graph to recursively find ways to recover the set of target values given the set of available values. If there is more that one way to restore a value, we choose the one with the smallest cost. The search result is a subgraph of the value search graph which we call a *route graph*. For any path, a route graph shows a specific way to recover each target value from available values. Finally, the forward and reverse functions are built from a route graph. Figure 2 illustrates this process.

4.2 Building the Value Search Graph

We first build an SSA graph for the target function. An SSA graph [3,10], built based on SSA form, consists of vertices representing operators, function symbols, or ϕ functions, and directed edges connecting uses to definitions of values. It shows data dependencies between different variables. The full algorithm for building an SSA graph is presented in [19]. Figure 3(a)(b) show the SSA-transformed CFG and its SSA graph for the function in Figure 1(a). In this example, `a` and `b` are two target variables with initial values $\mathtt{a}_0$ and $\mathtt{b}_0$, and final values $\mathtt{a}_3$ and $\mathtt{b}_2$.

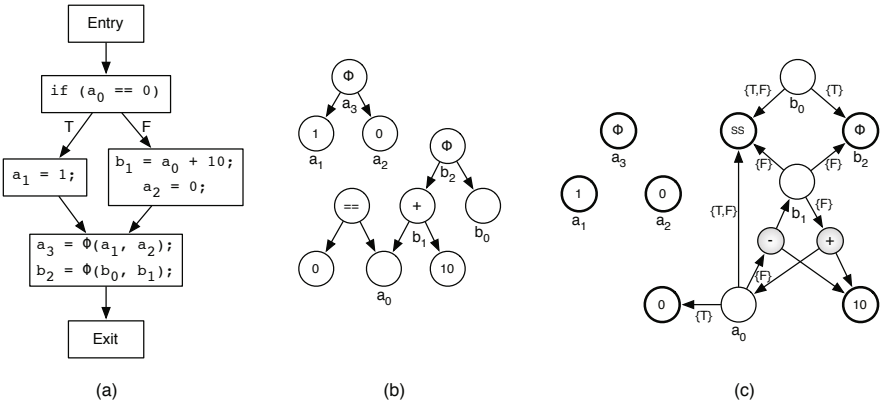


Fig. 3. (a) The SSA-transformed CFG of the function in Figure 1(a) (b) The corresponding SSA graph (c) The corresponding value search graph. Nodes with bold outlines are available nodes; outgoing edges for these nodes are omitted because available nodes need not be recovered. 'SS' is the special state saving node. Edges are annotated with their CFG path set.

A *value search graph* enables efficient recovery of values by explicitly representing equality relationships between values. Unlike an SSA graph, operation nodes are separated from value nodes in the value search graph, since their treatment is different for recovering values. An edge connecting two value nodes u and v implies that u and v have the same value. An edge from value node u to an operation node *op* means that u is equal to the result of evaluating *op* with its operands. To recover the value associated with node v, we can recursively search the graph starting at v.

We attach a set of CFG paths to each edge in a value search graph, meaning the edge is applicable only if one of the CFG paths in that set is selected in the original function. For operation nodes in the SSA graph, let the set of paths attached to each outgoing edge be the CFG paths for which the corresponding operation is executed. Similarly, for ϕ nodes, each reaching-definition edge should be annotated with all CFG paths for which the corresponding reaching definition reaches the ϕ function. We will describe an implementation of the path set representation later.

During the execution of the forward function, once a variable is assigned with a new value, its previous value may be destroyed and cannot be retrieved. To guarantee that a search in the value search graph can always restore a value, we introduce special *state saving edges*. The idea behind these edges is that each value may be recovered by storing it during the forward execution. Whenever a state saving edge appears in the search results, the forward function is instrumented to save the corresponding value. The path set associated with a state saving edge for a value node v is the set of all paths that include v's definition. All state saving edges point to a unique *state saving node*.

We apply the following rules to convert an SSA graph into a value search graph:

- For simple assignment $v = w$, there is a directed edge from v to w in the SSA graph. Since we can retrieve w from v, add another directed edge from w to v with the same path set.
- A ϕ node in the SSA graph has several outgoing edges connecting all its possible definitions. For each of those edges, add an opposite edge with the same path set.
- For each operation node in the SSA graph, split it into an operation node and a value node, with an edge from the value node to the new operation node. The new operation node takes over all outgoing edges, and the value node takes over all incoming edges.
- If an equality operation (`==`) is used as a branching predicate and its outcome is true, we know that the two operands are equal. Therefore, we add edges from each operand to the other, with a path set for the edge equal to the path set of the *true* CFG edge out of the branch. We add the edges analogously for a not-equal operation (`!=`), but with the path set from the *false* side of the branch.
- For every value that is not available, insert a state saving edge from the corresponding value node to the state saving node.

Lossless operations. For certain operations, such as integer addition and exclusive-or, we can recover the value of an operand given the operation result and the other operand. For example, if $a = b + c$, we can recover b given a and c. For each such lossless operation, insert new operation nodes that connect its result to its operands, allowing the operands to be recovered from the result. The new nodes are added according to the following rules:

- Negation (`a = -b`) and bitwise not (`a = ~b`): the new operations are `b = -a` or `b = ~a`, respectively.
- Increment (`++a`) and decrement (`--a`): insert `--a` or `++a`, respectively.
- Integer addition (`a = b + c`) and subtraction: for addition, the new operations are `b = a - c` and `c = a - b`; analogously for subtraction.
- Bitwise exclusive-or (`a = b ^ c`): insert `b = a ^ c` and `c = a ^ b`

There are two special types of nodes in a value search graph: *target nodes* are value nodes containing target values, and *available nodes* are value nodes containing available values plus the state saving node. As an optimization, we never

create any outgoing edges for an available node. Figure 3(c) shows the value search graph built for the code in Figure 1(a). The available nodes are shown with a bold outline. Since the function only has two paths, we use labels 'T' and 'F' to represent the CFG paths passing through the true and false body in the target function, respectively. The '–' operation node connecting a_0 to b_1 and the constant value '10' is generated from the '+' operation. The edge from a_0 to '0' for the path 'T' is added based on the fact that $a_0 = 0$ on that path. The 'SS' node in the graph is the state saving node, and all unavailable nodes are connected to it. From the value search graph, we can find two valid ways to restore b_0 for the path 'T': b_0 to SS node and b_0 to b_2. Obviously the second one is better since it avoids a state saving operation, and this better selection will be produced from the search algorithm described later.

4.3 The Route Graph

A *route graph* is a subgraph of a value search graph connecting all target nodes to available nodes. Each route graph represents one way to restore the target values, and there may exist many valid route graphs for the same set of target values. Edges in the route graph may have different path sets than the corresponding edges in the value search graph. For each edge e in a route graph, let $P(e)$ denote the set of CFG paths that the edge is annotated with. The following properties guarantee that the route graph properly restores all target values:

I) Let $\mathcal{U}$ be the set of all CFG paths. Then, for each target node t,

$$\bigcup_{out \in \mathrm{OutEdges}(t)} P(out) \quad = \quad \mathcal{U}$$

II) For each node n that is neither a target node nor an available node,

$$\bigcup_{out \in \mathrm{OutEdges}(n)} P(out) = \bigcup_{in \in \mathrm{InEdges}(n)} P(in)$$

III) For each value node n, given any two outgoing edges $n \to p$ and $n \to q$, $P(n \to p) \cap P(n \to q) = \emptyset$

IV) If e is a route graph edge and its corresponding edge in the value search graph is e', then $P(e) \subseteq P(e')$

V) For each directed cycle with edges $e_1 \ldots e_n$, $\quad \bigcup_{i=1}^{n} P(e_i) = \emptyset$

Property I specifies that each target value is recovered for every CFG path. Property II means that each value is recovered exactly for the paths for which it is needed. Property III requires that for each CFG path, there is at most one way to recover a value. Property IV requires that the set of CFG paths associated with an edge in the route graph is a subset of the CFG paths originally associated with that edge in the value search graph. Finally, property V forbids self-dependence: restoring a value cannot require that value.

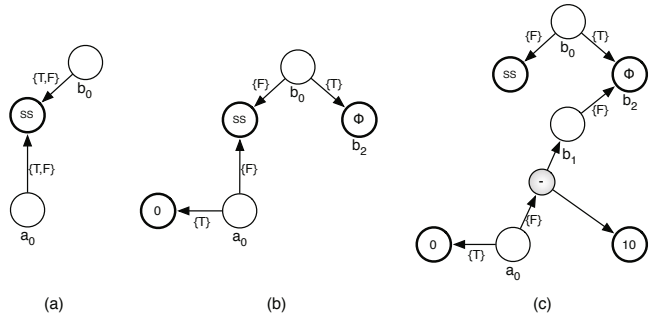


Fig. 4. Three different route graphs for the target values a_0 and b_0 given the the value search graph in Figure 3(c)

Figure 4 shows three valid route graphs for the value search graph in Figure 3. Route graph 4(a) only includes state saving edges. Route graph 4(b) takes advantage of the fact that for the 'T' path the values of both $\mathtt{a}_0$ and $\mathtt{b}_0$ are known; it only uses staving for the 'F' path. Route graph 4(c) improves upon route graph 4(b) by recomputing $\mathtt{a}_0$ as $\mathtt{b}_1\mathtt{-10}$ for the CFG path 'F'; state saving is only applied to $\mathtt{b}_0$ for path 'F'.

4.4 Searching the Value Search Graph

Costs in Route Graphs. As we have seen in Figure 4, there may be multiple valid route graphs that recover the target values, but with different overheads. In order to choose the route graph with the smallest overhead, we must define a cost metric.

Generally, there are two kinds of overhead in forward and reverse functions: execution speed and additional memory usage; we only consider the storage costs. State saving contributes the most to the overhead memory usage and it also significantly affects the running time of both forward and reverse functions. Storing the path taken during forward execution is the other factor that contributes to memory usage; this overhead is bounded and is the same for all route graphs, so we exclude it from our cost estimate. With each state saving edge in the value search graph, we associate a cost equal to the size of the value that must be saved; other edges have cost 0. The cost of a route graph for a specific CFG path is the sum of the cost of those edges whose annotated path sets include that CFG path.

In Figure 4, suppose the cost to store and restore either `a` or `b` is c, the following table shows the cost of three route graphs for each CFG path. Obviously the third route graph is the best one.

CFG path	route graph (a)	route graph (b)	route graph (c)
T	$2c$	0	0
F	$2c$	$2c$	c

We have defined the cost of a single CFG path; however, a route graph may have different costs for different CFG paths. When searching the value search

graph, we would like to treat groups of CFG paths that share some edges in the route graph together, rather than performing a full search for each CFG path. For this reason, the search algorithm partitions the CFG paths into disjoint sets of paths that have equal cost and we save the cost for each set of paths independently. In our search algorithm, we denote the costs of a route graph r as r.costSet.

$$r.\text{costSet} = \{\langle P_i, c_i\rangle | P_i \text{ is a set of CFG paths and } c_i \text{ is the cost}\}$$

Search Algorithm. Our search algorithm should aim to find a route graph that has the minimum cost for each path. Theoretically, however, searching for a minimal route graph is an NP-complete problem. To make the problem tractable, we apply the heuristic of finding a route graph for each target value individually; the individual route graphs are then merged into a route graph that restores all the target values. Similarly, in order to recover the value of a binary operation node, we recover each of the two operands independently and then combine the results.

The pseudocode for our heuristic search algorithm is presented in Algorithm 1. The `SearchSubRoute` function returns a route graph given a target node, the paths for which that node must be restored, and the set of value nodes visited so far. The algorithm explores all ways to recover the current node by calling itself recursively on all the nodes that are directly reachable from the current node; available nodes are the base case. Lines 5–10 handle recovering the values of operation nodes. In order to recover the value of an operation node, each of its operands must be recovered. Lines 11–14 return a trivial route graph for available nodes, with a cost of 0. The remaining body of the algorithm (lines 15–27) handles recovering a value node that is not available. Each of the out-edges of the target node may be used to recover its value for the CFG paths associated with that edge; these edges are explored in the `for`-loop in lines 15–23. The variable `newPaths` on line 17 represents the set of paths that we are both interested in and are associated with the current edge. In line 19, we recursively find a route graph that recovers the target value by recovering the target of the current outgoing edge. Lines 21–22 update the cost sets of the new route graph; if it provides a lower cost for some CFG path than the solutions found so far, the partial results are modified so that each CFG path is restored with the cheapest route graph. Finally, the route graph from line 19 is added to the list of partial results (line 23). After all out-edges of the target node have been explored, the partial results are merged into a single route graph and returned (lines 24–27). Note that it is unnecessary to check whether the target node has been successfully recovered, since the state saving edge always provides a valid route graph for the node. Figure 4(c) shows the route graph produced by the algorithm when searching the value search graph from Figure 3(c).

The search algorithm enforces properties I–IV from section 4.3 during its execution. To make sure that the search result does not contain cycles (property V), we record which value nodes are already in the route using a set `visited` in Algorithm 1. This alone is not sufficient to guarantee that the result is acyclic,

Algorithm 1. Searching for a route graph in a value search graph

Initial input: The search start point target, with paths = ∅, visited = ∅

```
1  SearchSubRoute(target, paths, visited)
2  begin
3      resultRoute ← ∅, subRoutes ← ∅
4      if target is an operation node then
5          foreach edge ∈ OutEdges(target) do
6              if edge.target ∈ visited then return ∅
7              newRoute ← SearchSubRoute(edge.target, paths, visited)
8              if newRoute = ∅ then return ∅
9              add edge and newRoute to resultRoute
10         return resultRoute
11     if target is available then
12         add target to resultRoute
13         add ⟨paths, 0⟩ to resultRoute.costSet
14         return resultRoute
15     foreach edge ∈ OutEdges(target) do
16         if edge.target ∈ visited then continue
17         newPaths ← edge.pathSet ∩ paths
18         if newPaths = ∅ then continue
19         newRoute ← SearchSubRoute(edge.target, newPaths, visited ∪ {target})
20         add edge with paths newPaths to newRoute
21         foreach ⟨paths, cost⟩ in newRoute.costSet do cost += edge.cost
22         foreach route in subRoutes do ChooseMinimalCosts(route, newRoute)
23         add newRoute to subRoutes
24     add target to resultRoute
25     foreach route in subRoutes do
26         if route.pathSet ≠ ∅ then add route to resultRoute
27     return resultRoute

28 ChooseMinimalCosts(route1, route2)
29 begin
30     if route1.pathSet ∩ route2.pathSet = ∅ then return
31     foreach ⟨paths1, cost1⟩ in route1.costSet do
32         foreach ⟨paths2, cost2⟩ in route2.costSet do
33             if paths1 ∩ paths2 = ∅ then continue
34             if cost1 > cost2 then
35                 paths1 ← paths1 − paths2
36                 Remove (paths1 ∩ paths2) from all edges of route1
37             else
38                 paths2 ← paths2 − paths1
39                 Remove (paths1 ∩ paths2) from all edges of route2
40     route1.pathSet = ⋃⟨paths,cost⟩∈route1.costSet paths
41     route2.pathSet = ⋃⟨paths,cost⟩∈route2.costSet paths
```

for there may be two different paths with identical cost to recover a single value node. If one way is chosen to recover a value node v during path of the search, and then later v is recovered differently for the same CFG path, a cycle may form. To prevent this situation from occurring, we always traverse out-edges in the same order of line 19 of Algorithm 1; the first route graph with the smallest cost is chosen. In addition, two paths coming from two different value nodes may also form a cycle when all costs on edges of the cycle are 0. We eliminate this possibility by replacing 0 by a small cost ε.

4.5 Instrumentation and Code Generation

Representing CFG Path Sets. Our search algorithm relies on efficiently computing intersection, union, and complement of CFG path sets, as well as testing whether the set of paths is empty; for this reason we suggest implementing the set representations as bit vectors. Ball and Larus [5] present an path profiling method in which each path is given a number from 0 to $m-1$, where m is the count of the CFG paths. We use their algorithm to number each path, and for each path we associate exactly one bit in the bit vector used to represent a path set.

Recording CFG Paths. We need to store path information in a way that allows us to efficiently record the CFG path taken (for forward execution), and to efficiently check if the path matches a given set of CFG paths attached to a route graph edge (for reverse execution). However, if we encode each path using its path number, then examining whether a path is a member of a set is inefficient. Instead we use a bit vector to record the CFG path, in which each bit represents the outcome of a branching statement. Since this method is similar to *bit tracing*[4], we call this bit vector a *trace*. Note that two branches may share the same bit if they cannot appear in the same path. Thus, the number of bits required to store the path taken is equal to the largest number of branches that appear on a single CFG path. Algorithm 2 calculates bit-vector position for each branch node accordingly.

In the forward function, we use an integer as the bit vector to record all predicate results[3]. Let `trace` be the variable recording a trace, initialized to zero; then the true edge of each branch node v is instrumented with the statement [4]

$$\texttt{trace = trace | (1 <<} \ \mathit{position}(v)\texttt{);}$$

where $\mathit{position}(v)$ is calculated by Algorithm 2. The variable `trace` is stored at the end of the forward function and restored at the beginning of the reverse function. Note that we can further optimize the instrumentation by moving a trace updating operations downward through the CFG and merging them.

[3] Potentially we could omit recording predicates that do not affect the reverse function.

[4] We use several operators in C/C++ syntax here and below, which includes bitwise OR operator `|`, bitwise AND operator `&`, bitwise left shift operator `<<`, equal to operator `==`, and logical OR operator `||` .

Algorithm 2. Generating the bit position for each branch node

```
foreach CFG node u in reverse topological order do
    if u is a leaf node then
        position(u) ← -1
    else if u is a branch node then
        /* u → v and u → w are its two out-going edges                    */
        position(u) ← max(position(v), position(w)) + 1
    else
        /* u → v is its out-going edge                                    */
        position(u) ← position(v)
```

In the reverse function, we must test if `trace` matches the path sets that appear on route graph edges. We start with transforming each path in the set into a trace (the trace for each path can be computed by the same means as recording a trace in the forward function). Then, checking if a path set contains a path represented by `trace` is done by comparing it to each trace. Suppose a path set containing two paths is transformed into two traces `01101` and `01001`. Instead of comparing `trace` to each of them as:

```
if (trace == 01101 || trace == 01001)
```

we can simplify this predicate by using a mask `11011` on `trace`:

```
if ((trace & 11011) == 01001)
```

The combined trace for `01101` and `01001` is `01`$\times$`01`, where $\times$ denotes that the bit does not matter. Given a set of traces, we can combine pairs repeatedly to reduce the size of the set. This greatly reduces the complexity of the branching statements in the reverse code.

Algorithm 3 starts out with all traces corresponding to a set of CFG paths and merges them into a minimal set of traces that can be used to test membership in the set. The intuition behind Algorithm 3 is that if the traces are sorted so that bit i is the least significant bit, the traces that are identical to each other except for bit i will be adjacent. However, if we are careful we don't have to pay the full sorting cost for each bit i. If the traces are sorted when their bits are considered in the order $b_1 b_2 \ldots b_{i-1} \quad b_k b_{k-1} \ldots b_i$ and we want to sort them according to the bit order $b_1 b_2 \ldots b_{i-2} \quad b_k b_{k-1} \ldots b_{i-1}$, we need only sort each sequence of the trace for which bits 1 through $(i-2)$ are identical. For each such sequence, there are at most three sorted subsequences, indexed by bit b_{i-1}; these can be merged in linear time (similarly to mergesort). If we use a linear-time sort, such as radix sort, for the first iteration, the overall runtime of Algorithm 3 is $O(kn)$, where n is the size of the path set.

After the merge, if we have n traces $t_1, ..., t_n$ for a path set, the resulting predicate would be:

`if ((trace &` $mask_1$`) ==` obj_i `|| ... || (trace &` $mask_n$`) ==` obj_n`)`

Algorithm 3. Merging a set of path traces

```
MergePathTraces(traces)
begin
    /* Each trace has k "bits", and each bit is 0, 1, or ×          */
    /* Bits are numbered ascendingly; e.g. m = b1 b2 ... bk          */
    for i ← k down to 1 do
        /* Note: for i = k, the bit ordering is m = b1 b2 ... bk    */
        Sort traces, where trace bits are ordered b1 b2 ... bi−1  bk bk−1 ... bi
        for j ← 2 to Length(traces) do
            if traces[j − 1] and traces[j] match except for bit i then
                set bit i to × for traces[j − 1]
                delete traces[j]
```

For each trace t_i, $mask_i$ is obtained by setting all bits which are $\times$ in t_i to 0 and others to 1, and obj_i equals $mask_i$ & t_i.

Inserting State Saving Statements. The other instrumentation in the forward function are state saving statements, which are inserted according to the state saving edges in the route graph. For each state saving edge in the route graph, suppose the variable to store is *var* and the path set on this edge is P. Our task is finding one or several locations to store *var* according to the path set P, ensuring that *var* is only saved once for each CFG path in P.

To find such locations, we first compute the corresponding path traces T of P from Algorithm 3. For each trace in T, we traverse the CFG from the entry. When we reach a branch node, check the corresponding bit in the trace: fall through the true edge if the bit is 1, false edge if the bit is 0. If the bit is $\times$, the traversal forks and that bit is assigned to 0 and 1 respectively forming two new traces; and for each concretized trace the descent continues. The descent stops immediately when all bits which are not checked in the trace are $\times$. After this process, we obtain one or more locations where the descent has stopped. In each location we find a point where the definition of *var* is reachable and a state saving statement is inserted there. However, it is possible that the path set containing the paths passing through this location is larger than the one on which the state saving is needed. In this case, we guard the state saving statement with a branch whose predicate corresponds to the trace at this location.

Building a CFG for the Reverse Function. We build the CFG for the reverse function from a route graph; the reverse CFG is acyclic and each path in it must obey the data dependencies represented in the route graph. Each outgoing edge from a value node in the route graph will be translated to a statement in the reverse function.

There could be a large number of correct reverse CFGs for a route graph, resulting in different control flows and different numbers of branches. We choose

to build a structured CFG to simplify the translation to source code. We also attempt to minimize the number of predicates in the CFG.

There are three kinds of statements that can be generated from a route graph:

- An operator node with its operands and result induces an operation statement, such as `a = b + c`.
- An edge with value nodes as both ends induces an assignment statement.
- An edge pointing to the SS node induces an value restoration statement.

The statements generated from route graph edges retain the path sets attached to the corresponding edges. We build basic blocks of statements that all share the same path sets, and insert branches so that each basic block is executed when the corresponding path is taken in the forward function. While enforcing the path set constraints ensures correct control flow, producing correct data flows depends on the order in which statements are inserted in the CFG. Note that a route graph corresponds to explicit data dependencies, and for each CFG path in the forward function it is acyclic due to property V from section 4.3. Hence, if we order statements in the reverse topological order of the route graph edges, dataflow dependencies are correctly maintained.

Algorithm 4 shows how to build a CFG for the reverse function. We keep a set of basic blocks, `openBlocks`, to which new statements can be appended. We also maintain a set of statements, `pendingStmts`, whose data dependencies have been satisfied, but which have not yet been inserted in the CFG. Each basic block has an associated path set; these are the paths in the forward function for which the corresponding basic block in the reverse function should execute. Similarly, each statement has a set of paths from the forward function. If there is a pending statement and an open basic block whose path sets match, we simply append the statement to the basic block. When a statement is inserted into the CFG, the data dependencies of new statements may now be satisfied; we call the function `BuildReadyStatements` to generate the statements that are now valid for insertion. If there is no pending statement whose path set matches the path set of an open basic block, we must insert or join a branch in the CFG. When a branch is inserted, two new basic blocks are created and the basic block containing the branch is closed. When a branch is joined, the joined basic blocks are closed and a new open basic block is created.

Note that it is possible that the instrumentation to the forward function brings additional implicit data dependencies. For example, if stack is used for state saving the order of values popped in the reverse function should be opposite of the order of pushes in the forward function. In this case, we can order those state saving statements in `pendingStmts` according to the order in which values are pushed.

Generating Code. The forward function is generated by copying the target function and adding state saving and control flow instrumentation (section 4.5). The reverse function is translated from the CFG built by Algorithm 4. Translating a structured CFG to source code is straightforward. Since each variable in the reverse CFG is in SSA form, we can use the versioned name during code

Algorithm 4. Generating a CFG for the reverse function from a route graph

```
GenerateReverseCFG(routeGraph)
begin
    cfg ← ∅, pendingStmts ← ∅, openBlocks ← ∅, pathSetPairs ← ∅
    foreach valNode in routeGraph do valNode.pathSet ← ∅
    foreach available node availNode in routeGraph do
        BuildReadyStatements(availNode, 𝒰, pendingStmts)
    cfg.entry ← BuildBasicBlock(𝒰)
    while pendingStmts ≠ ∅ do
        if ∃s ∈ pendingStmts, b ∈ openBlocks, and s.pathSet = b.pathSet then
            Append s to b
            valNode ← the source node of the edge that generated s
            BuildReadyStatements(valNode, s.pathSet, pendingStmts)
        else if ∃s ∈ pendingStmts, b ∈ openBlocks, and s.pathSet ⊂ b.pathSet
        then
            Append to b a branch, with the predicate generated from s.pathSet
            b1 ← BuildBasicBlock(s.pathSet)
            Append s to b1
            b2 ← BuildBasicBlock(b.pathSet − s.pathSet)
            Insert into cfg edges from b to b1 and b2 with labels true and false
            Add ⟨b1.pathSet, b2.pathSet⟩ to pathSetPairs
            openBlocks ← openBlocks − {b}
            valNode ← the source node of the edge that generated s
            BuildReadyStatements(valNode, s.pathSet, pendingStmts)
        else if ∃b1, b2 ∈ openBlocks, and ⟨b1.pathSet, b2.pathSet⟩ ∈ pathSetPairs
        then
            b ← BuildBasicBlock(b1.pathSet ∪ b2.pathSet)
            Insert into cfg two edges, from b1 and b2 to b
            pathSetPairs ← pathSetPairs − {⟨b1.pathSet, b2.pathSet⟩}
            openBlocks ← openBlocks − {b1, b2}
            if |openBlocks| = 1 then break
    return cfg

BuildReadyStatements(valNode, nodeAvailablePaths, pendingStmts)
begin
    valNode.pathSet ← valNode.pathSet ∪ nodeAvailablePaths
    foreach edge ∈ InEdges(valNode) do
        if edge.pathSet ⊆ valNode.pathSet then
            if edge.source is an operation node then
                Set edge to be a available for edge.source
                if all operands of edge.source are available then
                    Add to pendingStmts the statement for for edge.source, with
                    path set edge.pathSet
            else
                Add to pendingStmts the statement for edge, with path set
                edge.pathSet

BuildBasicBlock(pathSet) begin
    Build an empty basic block b and attach path sets pathSet to it.
    cfg ← cfg ∪ { b },    openBlocks ← openBlocks ∪ { b }
    return b
```

generation. Because our framework generates source code that is later compiled with another compiler, the redundant variables will be optimized away; the only drawback of this approach is readability. If readability is an issue, we can compute data dependencies in the reverse CFG and then remove versions attached to variables where this does not affect data dependencies. After version removal, we would also remove self-assignment statements such `a = a`. Figures 1(b) and 1(c) show the generated forward and reverse functions from the code in Figure 1(a).

5 Handling Loops

Our discussion of loops only considers natural loops with only one entry; loops with more than one entry are quite rare in practice and can be transformed into natural loops [19]. Consider a loop that takes n iterations and modifies a value. There are two approaches to restoring that value. The first one is generating another loop in the reverse function which contains the inverse of the loop body and also executes n times. The other approach is forgoing generating a loop and using other methods such as state saving. We refer to the first approach as the loop solution, and to the second as the non-loop solution.

Although the loop solution may be able to restore a value without state saving, there are two pitfalls with this approach. First, a natural loop only has one entry, but may have several exits (e.g. `break` and `return`). The loop exits may point to anywhere topologically after the loop, so the inverse of a loop body may have several entries with varying reaching definitions. Second, recovering a value through reverse iterations may be less efficient than state saving. Memory storage inside a loop, either for state saving or recording control flow, is multiplied by the number of iterations. Even without memory overheads inside a loop, it may be faster to just save and restore a value than to recompute it through a long iteration.

In this paper we only deal with scalar variables, which seem unlikely to benefit from the loop solution. For loops containing arrays and function calls, the loop solution may be better or even necessary. Space restrictions preclude describing both methods. We will describe the non-loop solution below and show the brief idea of the loop solution.

Non-loop solution. In SSA form each variable has only one definition, but if that definition is in a loop, the versioned variable no longer represents a single value; the algorithm from section 4 is not applicable directly. Our non-loop solution is removing value nodes containing definitions in loops (including the definition from a ϕ function in the loop header) when building the value search graph. Besides, each loop is reduced into a single node, and the loop-free algorithm applies.

Loop solution. The loop solution applies a transformation on the loop to make it only have one exit. This is done by separating the last iteration from others, since it is the only iteration that may exit the loop. Afterwards, the loop

header becomes both the entry and exit of each iteration, and the loop as a region becomes a hammock [12]. We then build the value search graph for this hammock and embed it to the value search graph of the whole method. The search algorithm still determines how to restore each value according to the cost on edges.

6 Experiment Results

We have implemented the framework in our C/C++ source-to-source translator Backstroke based on the ROSE compiler. Since this paper focuses on arbitrary control flows and basic operations with only scalar data types, instead of trying to reverse real-world code, which usually includes function calls, non-scalar data types, aliasing, etc., we employ some representative synthetic benchmarks to illustrate the power of our algorithm. Those benchmarks are listed below.

- **NoBranch**: A variable is modified in the function.
- **Branches1**: There are many CFG paths in the function and only one variable is modified on one path.
- **Branches2**: There are many CFG paths in the function and on each path a distinct variable is modified.
- **Branches3**: There are many CFG paths in the function and a variable is modified up to three times on some paths and is not modified on other paths.
- **Loop1**: A loop in which a variable is modified. The loop is intended to have many iterations at runtime.
- **Loop2**: A loop containing a simple branch and two variables are modified in the true and false body respectively. The loop is intended to have many iterations at runtime.

In addition, each benchmark has two versions in which every variable is modified differently: in the first one, each variable is modified by an assignment; the other one modifies each variable using an increment operation (`++`) so that the assignment can be reversed trivially. We denote those two versions by **Assignment** and **Increment**.

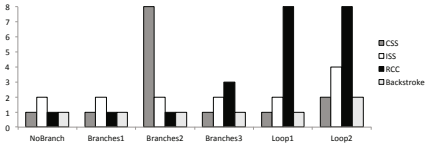
We compare our method[5] to three other approaches commonly employed in the OPDES community to implement rollback:

- **CSS**: Copy state saving. Every target variable is stored at the beginning of the forward function and restored in the reverse function. Here we only store the variables that are potentially modified.
- **ISS**: Incremental state saving. A variable is stored only the first time it is modified. This technique is traditionally implemented by storing the variable's address along with its value, so one can check if the variable is already stored.
- **RCC**: Reverse C compiler [8] is a syntax-directed incremental inversion translator (see section 2).

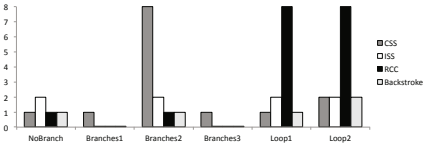
[5] Note that for loops we use the non-loop solution as defined in section 5.

We count the maximum and minimum memory used for state saving. The memory used to record the control flows outside of loops (including the counter recording the number of iterations in a loop) is ignored because it does not scale with the size of the program state. Figure 5 shows the experiment results, in which (a) and (b) are maximum and minimum memory usage for all benchmarks of the **Assignment** version, and (c) and (d) are of the **Increment** version. The height of each column represents the memory usage.

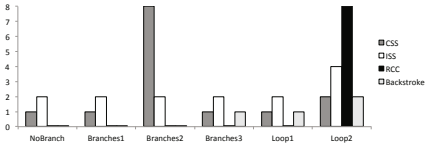
From the result we can see for most benchmarks Backstroke is the most efficient, which is because our method integrates the advantages from both incremental reverse execution and incremental state saving. **ISS** stores the address of every variable which introduces a large overhead if the address's size is comparable to that of a value's (for scalar data) meanwhile we utilize the CFG path to ensure each variable is stored only once, with much less overhead. **ISS** outperforms **CSS** when there are many variables which are potentially modified but only a small number of them are modified during each execution (see Figure 5 **Branches2**). That is why incremental state saving performs very well when each event only modifies a small portion of the whole state.



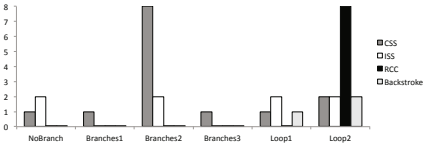
(a) Maximum memory usage of **Assignment**



(b) Minimum memory usage of **Assignment**



(c) Maximum memory usage of **Increment**



(d) Minimum memory usage of **Increment**

Fig. 5. Experiment results

From comparing the results from Figure 5 (a)(b) and (c)(d), it is clear that the reverse execution approaches can save much memory. But the amount of benefit from reverse execution is determined by the number of opportunities for reverse computation. For programs that do not have many lossless operations such as `++` and `+=`, state saving still plays an important role in their inversions.

We must be very cautious when reversing a loop. If the loop solution is applied, we have to determine if storing control flow information is worth it or not. The result of **Loop2** from **RCC** shows that if the number of iterations is large,

storing control flows is not good idea. Saving state inside a loop normally is not a wise choice, as the result of **Loop1** + **Assignment** from **RCC** show.

7 Conclusion and Future Work

We have shown a novel framework for program inversion that combines the advantage of both incremental state saving and incremental reverse execution. Powered by compiler analysis and taking the cost for each operation into account, we gain global optimization through a search algorithm that makes it possible to find the best strategy when reversing a program. The experiment results show that our method is effective comparing to other well-known methods.

Our future work will target reversing functions with structures, arrays, function calls, and aliasing. We believe that our framework is general enough to be reused to solve those problems. For example, a function call can be treated as a special operation. We can extend the value search graph to take advantage of SSA extensions for pointers, arrays, object access, and function calls [9,13,18], allowing us to integrate them into our inversion algorithm. We would also like to run our translator on real-world large simulations and measure the performance gains from parallel execution.

References

1. Abramov, S., Glück, R.: The Universal Resolving Algorithm: Inverse Computation in a Functional Language. Science of Computer Programming 43(2-3), 193–229 (2002)
2. Akgul, T., Mooney III, V.J.: Assembly instruction level reverse execution for debugging. ACM Transactions on Software Engineering and Methodology 13(2), 149–198 (2004)
3. Alpern, B., Wegman, M.N., Kenneth, F.: Detecting Equality of Variables in Programs. In: PPL (January 1988)
4. Ball, T., Larus, J.R.: Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems 16(4), 1319–1360 (1994)
5. Ball, T., Larus, J.R.: Efficient Path Profiling. In: MICRO 1996, pp. 46–57 (1996)
6. Biswas, B., Mall, R.: Reverse execution of programs. ACM SIGPLAN Notices 34(4), 61–69 (1999)
7. Briggs, J.S.: Generating reversible programs. Software: Practice and Experience 17(7), 439–453 (1987)
8. Carothers, C.D., Perumalla, K.S., Fujimoto, R.M.: Efficient Optimistic Parallel Simulations Using Reverse Computation. In: PADS (1999)
9. Chow, F., Chan, S., Liu, S.-M., Lo, R., Streich, M.: Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 253–267. Springer, Heidelberg (1996)
10. Cooper, K.D., Taylor Simpson, L., Vick, C.A.: Operator strength reduction. ACM Transactions on Programming Languages and Systems 23(5), 603–625 (2001)
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)

12. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (1987)
13. Fink, S., Knobe, K., Sarkar, V.: Unified Analysis of Array and Object References in Strongly Typed Languages. In: SAS 2000. LNCS, vol. 1824, pp. 155–174. Springer, Heidelberg (2000)
14. Fujimoto, R.M.: Parallel and Distributed Simulation Systems. Wiley (2000)
15. Glück, R., Kawabe, M.: Revisiting an automatic program inverter for Lisp. ACM SIGPLAN Notices 40(5), 8–17 (2005)
16. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems 7(3), 404–425 (1985)
17. Kawabe, M., Glück, R.: The Program Inverter LRinv and Its Structure. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 219–234. Springer, Heidelberg (2005)
18. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: POPL 1998, pp. 107–120. ACM Press, New York (1998)
19. Muchnick, S.S.: Advanced Compiler Design and Implementation (1997)
20. Ross, B.J.: Running programs backwards: The logical inversion of imperative computation. Formal Aspects of Computing 9(3), 331–348 (1997)
21. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S.: Path-based inductive synthesis for program inversion. In: PLDI 2011, p. 492. ACM Press, New York (2011)