AUTOMATED SYNTHESIS FOR PROGRAM INVERSION

A Thesis
Presented to
The Academic Faculty

by

Cong Hou

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2013

AUTOMATED SYNTHESIS FOR PROGRAM INVERSION

Approved by:

Doctor Richard Vuduc, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Doctor Richard Fujimoto
School of Computational Science and Engineering
Georgia Institute of Technology

Doctor Santosh Pande
School of Computer Science
Georgia Institute of Technology

Doctor Daniel Quinlan
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Doctor David Jefferson
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Date Approved: May 17 2013

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my advisor and committee chair, Doctor Richard Vuduc, who continuously supports me throughout the last five years in Georgia Tech. Without his patience, immense knowledge, convincing guidance, and persistent help, this dissertation would not have been possible.

I would like to thank my committee members, Doctor Daniel Quinlan, Doctor David Jefferson, Doctor Richard Fujimoto, and Doctor Santosh Pande, whose work demonstrated to me the spirit of true scholarships. Especially a thank you to Doctor Daniel Quinlan and Doctor David Jefferson, who introduced me to the Backstroke project when I interned at LLNL, and whose tolerance encouraged me to keep working on my dissertation.

At last, I want to give special thanks to my dear and sweet wife, Ruoyan, for her support and understanding during my PhD study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In this thesis, we consider the problem of synthesizing program inverses for imperative languages. Specifically, we will build a compiler framework named *Backstroke* [28] that can generate a reversible program and an inverse program for a given input program, by employing several intermediate representations and program analyses.

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 [18]. In this context, the ability to perform rollback by running time- and space-efficient reverse programs, rather than saving and restoring large amounts of state, can make OPDES more practical. Synthesizing inverses also appears in numerous software engineering contexts, such as debugging, synthesizing undo code, or even generating decompressors automatically given only lossless compression code. We will give a motivating concrete example of reversible programs in Chapter II.

This dissertation mainly contains three chapters. In Chapter II, we focus on handling programs with only scalar data and arbitrary control flows. By building a value search graph (VSG) that represents recoverability relationships between variable values, we turn the problem of recovering previous values into a graph search problem. The VSG is built based on Static Single Assignment (SSA) [13]. Forward and reverse programs are generated according to the search results. For any 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).

In Chapter III we extend our method to handling programs containing arrays. Based on Array SSA [25], we develop a modified Array SSA from which we could easily build equalities between arrays and array elements. Specifically, to represent the equality between two arrays, we model array subregions explicitly. During the search those subregions will be calculated to guarantee that all array elements will be retrieved. We also develop a demand-driven method to retrieve array elements from a loop, in which each time we only try to retrieve an array element from an iteration if that element has not been modified in previous iterations. To ensure the correctness of each retrieval, the boundary conditions are created and checked at the entry and the exit of the loop.

In Chapter IV, we introduce several techniques of handling high-level constructs of C++ programs, including virtual functions, copying a C++ object, C++ STL containers, expressions with several side effects, inter-procedural function calls, etc. Since C++ is an object-oriented (OO) language, our discussion in this chapter can also be extended to other OO languages like Java.

1.1 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 $s_1 s_2 \ldots s_n$ are executed in the forward directions, the reverse function executes statements $s_n^{-1} \ldots s_2^{-1} s_1^{-1}$. The incremental approach cannot handle unstructured control flows and is difficult to apply in the presence of early returns from functions; the approach presented in this thesis suffers from neither of these shortcomings. Furthermore, 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

Syntax-directed approaches 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. It also provides the ability to record the control flows in the original program, so that in the reverse program the control flows can be reconstructed. An early example of syntax-directed incremental inversion is Brigg's Pascal inverter [8]. This approach was later extended to C and applied both to optimistic discrete event simulation [10] and reversible debugging [6]. Because this approach does not include any program analysis, the produced result is far from optimized. It also has many restrictions. For example, it cannot handle unstructured programs, loops with early exits, and arrays, among other program constructs.

Consider the specific instance of the Reverse C Compiler (RCC) [10]. In their method, if a state variable is modified by a constructive operation like `++` or `+=`, then in the reverse program that state variable can be recovered without state saving. For example, if the state variable `s` is modified by `s += t`, then it can be recovered by `s -= t`. Otherwise, the state saving will be performed to that state variable before it is modified in the forward program. The RCC requires that all statements in the original program contain simple expressions, and that the program is structured. Because no program analysis is involved, the generated result may be far from optimized. For example, a state variable may be modified several times in the program, in which case one state saving is enough before its first modification to recover that value. But RCC may insert several state saving statements that consume more memory space. By contrast, in our approach we use program analysis to determine that this state variable is stored only once. In addition, RCC recovers every variable that is modified in the original program. This approach may lead to restoring many variables not actually related to state variables. By contrast, our method considers just the state variables. Additionally, we improve on RCC in that we can handle arbitrary control flows whereas RCC can only handle structured programs. This is because RCC relies on proper shapes of the original program to build the reverse program.

Value graph based approach Akgul and Mooney also previously improved on RCC and related methods. Their incremental inversion algorithm 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 = 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. For example, suppose that before `a` is modified, its previous definition is `a = b + c`. Then, we may retrieve its old value by redefining it using the same expression: `a = b + c`. Note that in OPDES, the variables we want to retrieve are normally the input of the program that do not have "previous" definitions. Therefore, this technique only works on those local variables which are used to retrieved state variables. Another technique is to examine all the uses of `a` and see if the value of `a` can be retrieved from any of its uses. For example, if `a` is used to defined another variable `b` by `b = a + c`, then we could retrieve `a` from `a = b - c`. Note that this technique only works for naturally invertible operations like plus and minus. 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. The specific program analysis builds a graph called modified value graph (MVG), from which the desired value may be retrieved by performing a particular search on the graph. In Akgul and Mooney's technique, control flows are handled using the ϕ-functions produced when computing the program's SSA form [12]. However, this approach cannot handle loops. Our approach is inspired from their method but 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. Furthermore, our method can take care of arbitrary control flows, including loops.

A related line of work is inverting programs that are injective, without using any state saving. Such work tends to focus on inverting functional programs [1, 15, 19]. Approaches to inverting imperative programs include translation to a logic language [24]. By contrast, our method uses no such translation, and is therefore can be applied on imperative languages.

Template based program synthesis framework The PINS framework for program inversion [27] is a template-based program synthesis framework. Rather than compiler transformations, as in our approach, PINS uses sketching and synthesis. In PINS, a programmer defines a template of the inverse with holes that a synthesizer attempts to fill in from a pool of candidates, using the underlying machinery of satisfiability (SMT) solvers. However, the current incarnation of PINS has weaknesses relative to our work. First, it is restricted to injective programs. Since we permit synthesis of a forward program, we can handle both injective and naturally non-injective programs. Secondly, it is only semi-automated, requiring both programmer annotations and templates. Our method is fully automated, relative to the limitations on aliasing alluded to previously. Thirdly, the synthesis time in PINS can be quite long and hard to predict, even for very small programs. Our method's transformation time is consistent with that of traditional compilation. Lastly, PINS does not guarantee a correctly synthesized inverse; one must apply a verification tool, such as a bounded model checker, to check the synthesized result. Our method produces correct inverses by construction. Collectively, these advantages make Backstroke more practical for production use than PINS.

1.2 Contributions

This thesis has mainly three contributions:

- In this thesis, we propose a novel automated method to generate forward and reverse programs for a given program. Most previous research on program inversion focuses on generating inverses from injective programs, but we can also handle non-injective programs by generating an injective version of it through storing necessary informations, which expands the scope of program inversion. By introducing a cost model to our method, we try to minimize the amount of information to be stored in the forward program.

- We developed a novel method to handle programs with arrays, with the help of array subregions and a modified version of Array SSA [25]. And the method we developed could also be used to handle object accesses by transforming them into array accesses.

- Our compiler can make use of human knowledge and generate better results. We feed those programmer-supplied information to our compiler and in practice the generated result are improved. We believe that based on this concept we could develop a framework or even a new language for program inversion for better results.

1.3 Limitations

Though Backstroke improves on prior work, it is not without limitations. Our method cannot handle programs with aliasing, and hence excludes some data structures like linked data structures. We will not apply any inter-procedural analysis in our method. For any function calls in the programs, we just assume there is no aliasing in the callee. In the last chapter, we will discuss several strategies to handle function calls. On the specific language level, in the last chapter we will discuss how to handle some C++ constructs, but Backstroke cannot handle all C++ language features. Those C++ features we cannot handle are listed below.

Aliasing Aliasing commonly exists in almost all imperative languages. In C++, aliases are brought by pointers and references, where specific data may be referenced by more than one pointers or references, and at compile time it is usually very difficult or even impossible to resolve all those aliases. In addition, the inter-procedural analysis is also heavily depending on aliasing analysis, as procedure calls usually have parameters passed by references.

Our method heavily depends on an intermediate representation called *Static Single Assignment (SSA)* [13], and it is difficult to build it for programs with aliasing. Our compiler also lacks of well formed aliasing analysis framework. Therefore, we will not discuss aliasing in this thesis and will assume there is no aliasing in the programs we handle.

Subset of C++ language We only handle a subset features of C++ language and those we cannot handle mainly include:

- Dynamic memory allocation and release.
- I/O and system calls.

- Exception handling.
- Template classes and functions.
- Function pointers.

In Chapter 2 and 3, we will restrict the target language to a subset of C++ language with following features:

- Each program is a C++ function with input and output variables which can be recognized by Backstroke.
- The type of each variable is either a basic scalar type (e.g. `int`, `float`, etc.), or an array type in which each element has a basic scalar type.
- Each scalar variable can only be modified by assignment operations. An array can only be modified by modifying one of its elements at a time also by assignment operations.
- All arithmetic operations and logical operations can be performed on variables or expressions.
- All control flow statements (except `throw`) can be used in the programs, including `if`, `else`, `switch`, `for`, `while`, `do while`, `goto`, `return`, `continue`, `break`.

In Chapter 4, we will discuss more constructs in C++ and extend our target language to including objects access, function calls, and polymorphism.

CHAPTER II

SYNTHESIS FOR PROGRAMS WITH ONLY SCALARS

In this chapter we will setup the program we are handling and introduce the framework of our method, which will also be used in the following chapters. As the first step, we will discuss how to handle programs with only scalars and arbitrary control flows. Two important intermediate representations will be introduces: a *value search graph* represents all equality relations in the program, and a *route graph* shows the data dependences in the reverse graph which is built as the search result on a value search graph. In addition, we will first consider loop-free programs, as its control flow paths are finite and much easier to represent. For programs with loops, we will treat each loop body as a subprogram so that we can apply the same approach to handling loop-free programs.

2.1 Handling loop-free programs

2.1.1 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*[1] M, producing $V' = \{v'_1 \dots v'_n\}$, the *final values* of the target 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}.

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

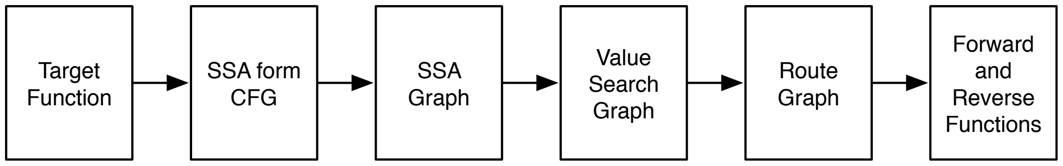


Figure 1: Overall framework of the inversion algorithm

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.

2.1.2 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 [12], 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.

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.

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, 11]. 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

```
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)

Figure 2: (a) The original function (b) The forward function (c) The reverse function

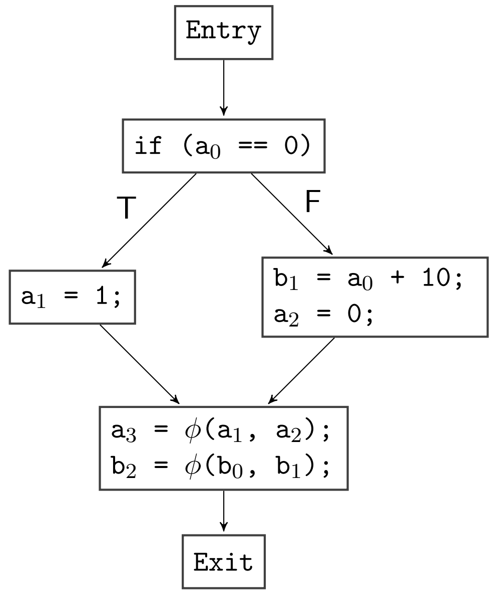
available values. Finally, the forward and reverse functions are built from a route graph. Figure 1 illustrates this process.

In this section, we will use the example shown in Figure 2 to illustrate our method.

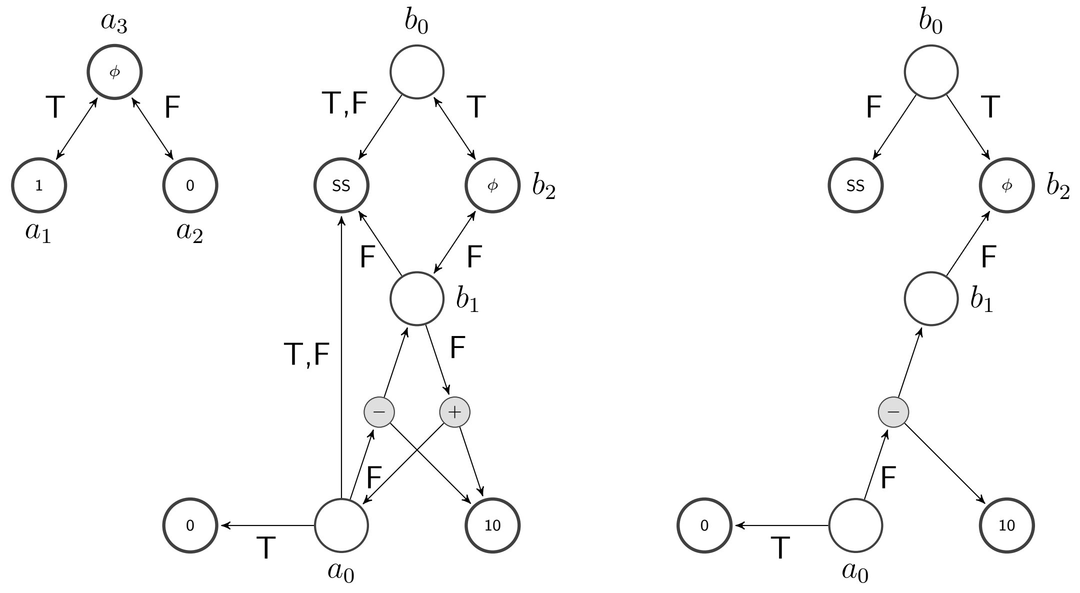
2.1.3 The value search graph (VSG)

We first build an SSA graph for the target function. An SSA graph [3, 11], 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 [20]. Figure 3(a)(b) show the SSA-transformed CFG and its SSA graph for the function in Figure 2(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 $\mathtt{b}_2$.

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



(a)



(b) (c)

Figure 3: (a) The SSA-transformed CFG of the function in Figure 2(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.

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.

More formally, a VSG is a graph containing two kinds of nodes: each *value node* represents a distinct value in the program; each *operation node* represents an operation and edges are connected with it and other value nodes, showing the equality between the result and the operation on the operands. For an edge between two value nodes, it shows those two nodes contain the identical value. Each edge will be attached with two informations: the condition of that equality and the cost to recover the value through this equality. We will discuss more details below.

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:

Operation name	Original operation	New operations added
Negation	`a = -b`	`b = -a`
Bitwise not	`a = ~b`	`b = ~a`
Logical not	`a = !b`	`b = !a`
Increment	`++a`	`--a`
Decrement	`--a`	`++a`

Integer addition	`a = b + c`	`b = a - c`
		`c = a - b`

Integer subtraction	`a = b - c`	`b = a + c`
		`c = b - a`

Bitwise exclusive-or	`a = b ^ c`	`b = a ^ c`
		`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 2(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.

2.1.4 The route graph (RG)

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,

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

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

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

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 $e_1 \dots e_n$, $\quad \bigcap_{i=1}^{n} P(e_i) = \emptyset$

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.

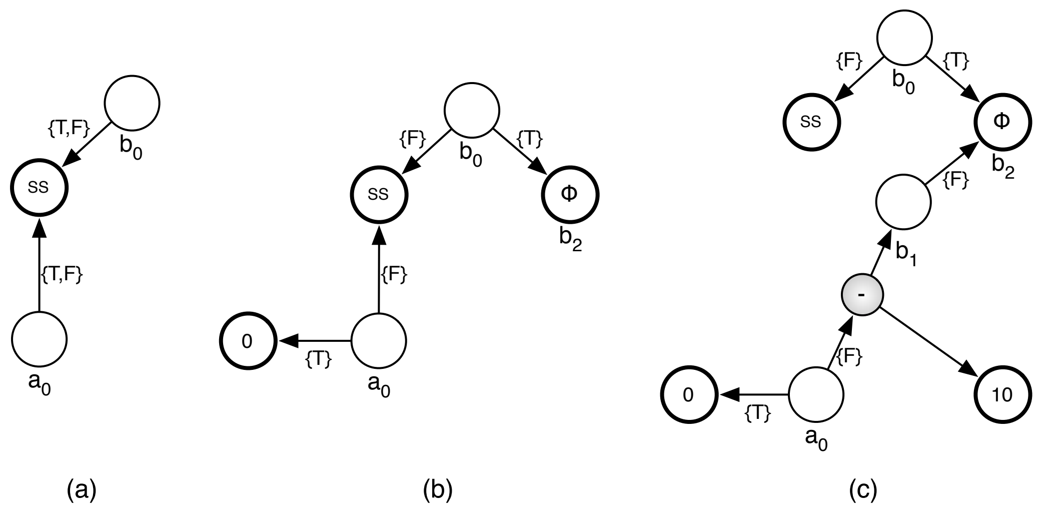


Figure 4: Three different route graphs for the target values a_0 and b_0 given the the value search graph in Figure 3(c).

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 $\mathtt{b}_1$`-10` for the CFG path 'F'; state saving is only applied to $\mathtt{b}_0$ for path 'F'.

2.1.5 Searching the value search graph

2.1.5.1 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.

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

2.1.5.2 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

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

```
Initial input: The search start point target, with paths = $\emptyset$, visited = $\emptyset$
SearchSubRoute(target, paths, visited)
begin
    resultRoute $\leftarrow \emptyset$, subRoutes $\leftarrow \emptyset$
    if target is an operation node then
        foreach edge $\in$ OutEdges(target) do
            if edge.target $\in$ visited then return $\emptyset$
            newRoute $\leftarrow$ SearchSubRoute(edge.target, paths, visited)
            if newRoute $= \emptyset$ then return $\emptyset$
            add edge and newRoute to resultRoute
        return resultRoute
    if target is available then
        add target to resultRoute
        add $\langle$paths, 0$\rangle$ to resultRoute.costSet
        return resultRoute
    foreach edge $\in$ OutEdges(target) do
        if edge.target $\in$ visited then continue
        newPaths $\leftarrow$ edge.pathSet $\cap$ paths
        if newPaths $= \emptyset$ then continue
        newRoute $\leftarrow$ SearchSubRoute(edge.target, newPaths, visited $\cup$ {target})
        add edge with paths newPaths to newRoute
        foreach $\langle$paths, cost$\rangle$ in newRoute.costSet do cost += edge.cost
        foreach route in subRoutes do ChooseMinimalCosts(route, newRoute)
        add newRoute to subRoutes
    add target to resultRoute
    foreach route in subRoutes do
        if route.pathSet $\neq \emptyset$ then add route to resultRoute
    return resultRoute

ChooseMinimalCosts(route1, route2)
begin
    if route1.pathSet $\cap$ route2.pathSet $= \emptyset$ then return
    foreach $\langle$paths1, cost1$\rangle$ in route1.costSet do
        foreach $\langle$paths2, cost2$\rangle$ in route2.costSet do
            if paths1 $\cap$ paths2 $= \emptyset$ then continue
            if cost1 $>$ cost2 then
                paths1 $\leftarrow$ paths1 $-$ paths2
                Remove (paths1 $\cap$ paths2) from all edges of route1
            else
                paths2 $\leftarrow$ paths2 $-$ paths1
                Remove (paths1 $\cap$ paths2) from all edges of route2
    route1.pathSet $= \bigcup_{\langle paths, cost\rangle \in route1.costSet}$ paths
    route2.pathSet $= \bigcup_{\langle paths, cost\rangle \in route2.costSet}$ paths
```

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 2.1.4 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, 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.

2.1.6 Instrumentation and Code generation

2.1.6.1 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.

2.1.6.2 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.

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 forward function, we use an integer as the bit vector to record all predicate results[2]. 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 [3]

`trace = trace | (1 <<` *position*(v)`);`

where *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.

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

[3]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 `||` .

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 $b_1b_2 \dots b_{i-1} \quad b_kb_{k-1} \dots b_i$ and we want to sort them according to the bit order $b_1b_2 \dots b_{i-2} \quad b_kb_{k-1} \dots b_{i-1}$, 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.

Algorithm 3: Merging a set of path traces

```
MergePathTraces(traces)
begin
    /* Each trace has k "bits", and each bit is 0, 1, or $\times$          */
    /* Bits are numbered ascendingly; e.g.  $m = b_1 b_2 \ldots b_k$          */
    for i ← k down to 1 do
        /* Note:  for i = k, the bit ordering is $m = b_1 b_2 \ldots b_k$     */
        Sort traces, where trace bits are ordered $b_1 b_2 \ldots b_{i-1} \quad b_k b_{k-1} \ldots b_i$
        for j ← 2 to Length(traces) do
            if traces[j − 1] and traces[j] match except for bit i then
                set bit i to $\times$ for traces[j − 1]
                delete traces[j]
```

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

$$\texttt{if ((trace \& } mask_1\texttt{) == } obj_i \texttt{ || ... || (trace \& } mask_n\texttt{) == } obj_n)$$

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.

2.1.6.3 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.

2.1.6.4 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 2.1.4. 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.

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

```
GenerateReverseCFG(routeGraph)
begin
    cfg ← $\emptyset$, pendingStmts ← $\emptyset$, openBlocks ← $\emptyset$, pathSetPairs ← $\emptyset$
    foreach valNode in routeGraph do  valNode.pathSet ← $\emptyset$
    foreach available node availNode in routeGraph do
        BuildReadyStatements(availNode, $\mathcal{U}$, pendingStmts)
    cfg.entry ← BuildBasicBlock($\mathcal{U}$)
    while pendingStmts $\neq \emptyset$ do
        if $\exists$s $\in$ pendingStmts, b $\in$ 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 $\exists$s $\in$ pendingStmts, b $\in$ openBlocks, and s.pathSet $\subset$ 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 $\exists$b1, b2 $\in$ openBlocks, and ⟨b1.pathSet, b2.pathSet⟩ $\in$ pathSetPairs then
            b ← BuildBasicBlock(b1.pathSet $\cup$ 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 $\cup$ nodeAvailablePaths
    foreach edge $\in$ InEdges(valNode) do
        if edge.pathSet $\subseteq$ 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 $\cup$ { b },     openBlocks ← openBlocks $\cup$ { b }
    return b
```

2.1.6.5 Generating code

The forward function is generated by copying the target function and adding state saving and control flow instrumentation (section 2.1.6.2). 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 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 2(b) and 2(c) show the generated forward and reverse functions from the code in Figure 2(a).

2.1.7 More examples

Except the example shown in Figures 2, here we show more examples including original program and forward & reverse programs.

In the first example, we show that when a variable is modified more than once on some control flow paths, we only store and restore it once correspondingly. In the code below, assume `s` is a state variable. There are two branches in the program, and `s` is modified in the true body in each branch:

```
void foo() {
    if (cond1)
        s = 0;
    if (cond2)
        s = 1;
}
```

A possible pair of forward and reverse programs are shown below:

```
void foo_forward() {
    int trace = 0;
    if (cond1) {
        trace |= 1;
        store(s);
        s = 0;
    }
    if (cond2) {
        trace |= 2;
        store(s);
        s = 1;
    }
    store(trace);
}
void foo_reverse() {
    int trace;
    restore(trace);
    if (trace & 2)
        restore(s);
    if (trace & 1)
        restore(s);
}
```

Note that in the forward program above, **s** will be stored twice on the control flow path passing through both true bodies. Backstroke can generate better result that avoids unnecessary stores, which are shown below.

```
void foo_forward() {
    int trace = 0;
    if (cond1) {
        trace |= 1;
        store(s);
        s = 0;
    }
    if (cond2) {
        trace |= 2;
        if (!(trace & 1))
            store(s);
        s = 1;
    }
    store(trace);
}
void foo_reverse() {
    int trace;
    restore(trace);
    if ((trace & 3) == 2)
        restore(s);
    if (trace & 1)
        restore(s);
}
```

In the second example, we show that the reverse program may have a different shape from that of the original program (RCC [10] always generates reverse programs with "reverse" shapes as the original programs). The original program is shown below with the state variables `s` and `t`:

```
void foo() {
    if (cond1) {
        if (cond2)
            return;
        ++s;
    }
    if (cond3)
        t = 0;
    else
        t = 1;
}
```

Note that if `cond1` and `cond2` are both true, there is no side effect. The forward and reverse programs generated by Backstroke are shown below:

```
void foo_forward() {
    int trace = 0;
    if (cond1) {
        trace |= 1;
        if (cond2) {
            trace |= 2;
            store(trace);
            return;
        }
        ++s;
    }
    store(t);
    if (cond3) {
        trace |= 4;
        t = 0;
    }
    else
        t = 1;
    store(trace);
}
void foo_reverse() {
    int trace;
    restore(trace);
    if ((trace & 3) != 3) {
        restore(t);
        if ((trace & 1) == 1)
            --s;
    }
}
```

Note that the recorded control flows should be stored at each exit of the program. The forward program has four control flow paths with three conditions but the reverse program only has three control flow paths with two conditions.

2.1.8 Comparison to other methods

We have implemented the framework in our C/C++ source-to-source translator Backstroke based on the ROSE compiler. Since this chapter 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[4] 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 [10] is a syntax-directed incremental inversion translator (see section 1.1).

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

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

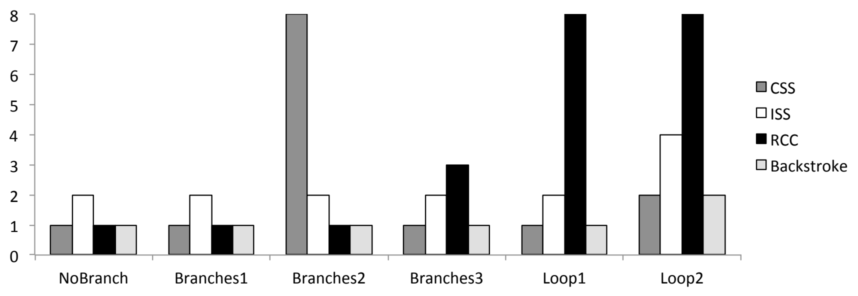
introduces a large overhead if the address's size is comparable to that of a value's (for scalar data) meanwhile we utilize the CFG path to ensure each variable is stored only once, with much less overhead. **ISS** outperforms **CSS** when there are many variables which are potentially modified but only 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.

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.

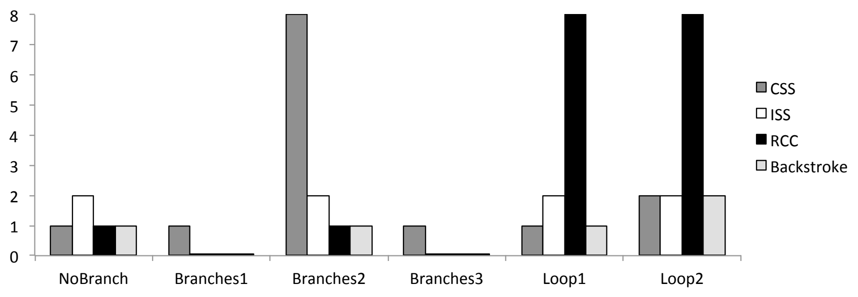
We must be very cautious when reversing a loop. If the loop solution is applied, we have to determine if storing control flow information is worth it or not. The result of **Loop2** from **RCC** shows that if the number of iterations is large, storing control flows is not good idea. Saving state inside a loop normally is not a wise choice, as the result of **Loop1** + **Assignment** from **RCC** show.

2.2 Handling programs with loops

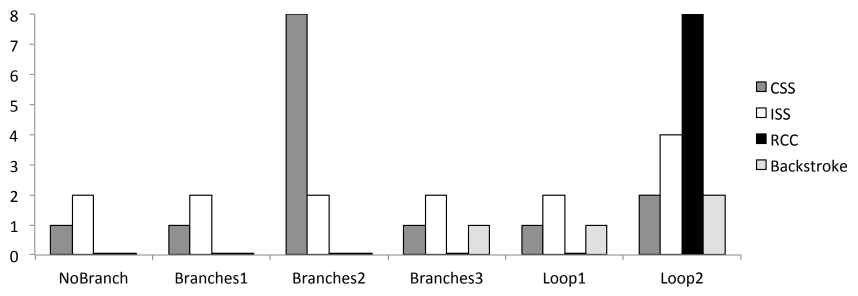
Unmodified, the method described above cannot handle loops for two key reasons. First, a loop results in cyclic paths in the CFG, whereas our prior analysis relies on paths being acyclic. Acyclic paths make it easy to check that the reverse program restores any desired input value no matter what path 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



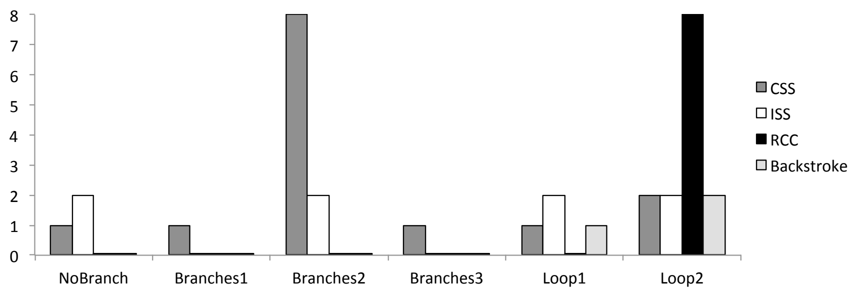
(a) Maximum memory usage of **Assignment**



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



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



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

Figure 5: Experiment results (y-axis shows memory usage normalized to CSS).

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 2.2.2 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).

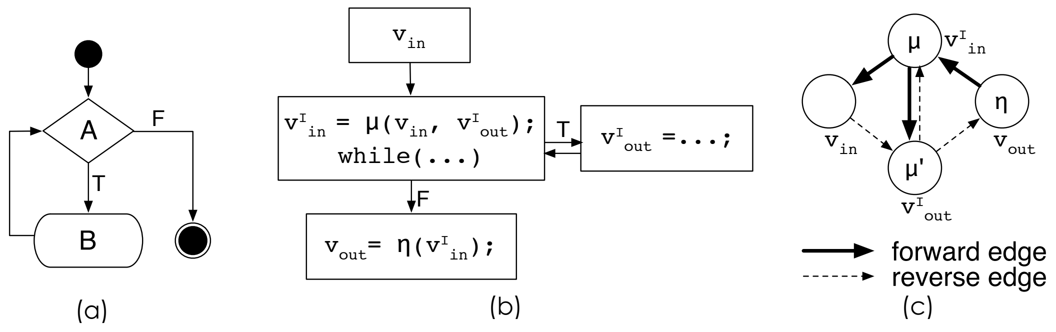


Figure 6: (a) The diagram of a while loop. (b) The CFG in loop-closed SSA form for a variable v modified in the loop. (c) Forward and reverse edges.

Because state saving in a loop is very expensive, we won't consider it in a loop. Moreover, it suffices for us to deal with a single loop without nested loops, which can be handled in the same way recursively.

2.2.1 Dealing with while loops

We first consider a while loop with the diagram shown in Figure 6(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* [26], illustrated in Figure 6(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 [23]. 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 6(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 $v_{\text{out}} = \eta(v^I_{in})$ 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^I_{in}, and v^I_{out}. This VSG result is shown in Figure 6(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^I_{in} and v^I_{out} 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 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 [17], 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 6(c), if the search reaches v^I_{in}, the algorithm begins two sub-searches beginning with v_{in} and v^I_{out}.
- 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.

2.2.1.1 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.

2.2.1.2 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* [29], 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 (first) and the generated loop in the reverse program (second) for the above example.

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

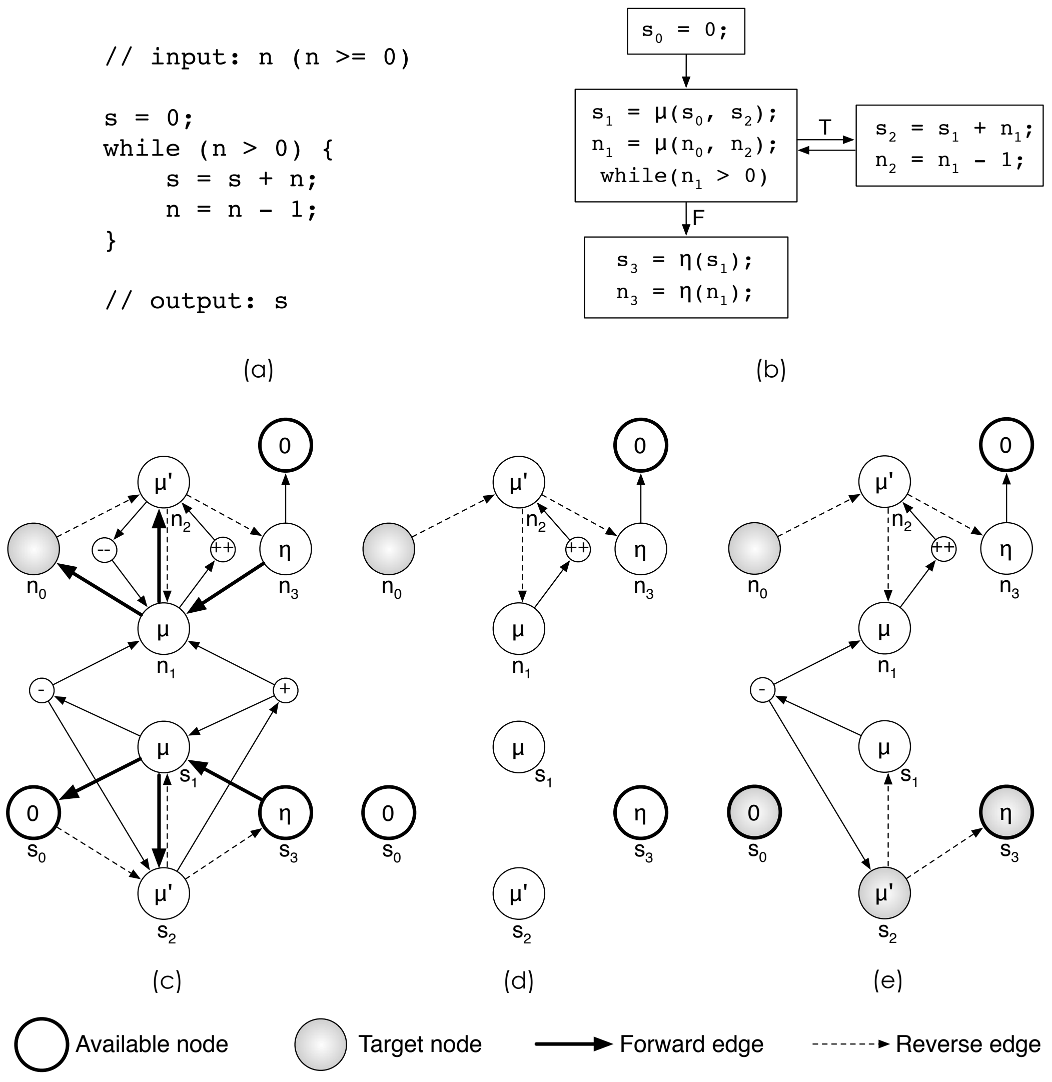
```
while (count > 0) {
    /* ... */
    count = count - 1;
}
```

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 7(a). Figure 7(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 7(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.[5] The search result for value n_0 is shown in Figure 7(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 7(e). As a result, we build the loop predicate from s and the reverse program is generated as below.

[5]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
```

```
s0 = 0;

s1 = μ(s0, s2);
n1 = μ(n0, n2);
while(n1 > 0)

s2 = s1 + n1;
n2 = n1 - 1;

s3 = η(s1);
n3 = η(n1);
```

Figure 7: (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;
}
```

Above we have built a reverse loop in the reverse program, but it is also possible that the reverse program contains a forward loop. For instance, if we change our example into the program shown below:

```
s = 0;
i = 0;
while (i < n) {
    i = i + 1;
    s = s + i;
}
```

Without modifying its semantic, its inverse will contain a forward loop which is shown below:

```
s2 = 0;
i = 0;
while (s2 != s) {
    i = i + 1;
    s2 = s2 + i;
}
n = i;
```

This is because the input of the new program n_0 is equal to i's final value, which is the output of the loop.

2.2.2 Dealing with loops other than while loops

In practice, the vast majority of loops have a single entry, which are called *natural loops* [20]. Loops with more than one entry are quite rare and can in fact be transformed into natural loops [20]. 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 8(a) shows a loop in a CFG, with a header (node 1) and two back edges ($4 \rightarrow 1$ and $5 \rightarrow 1$). There are two different exits from this loop, which are nodes 6 and 7. Figure 8(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

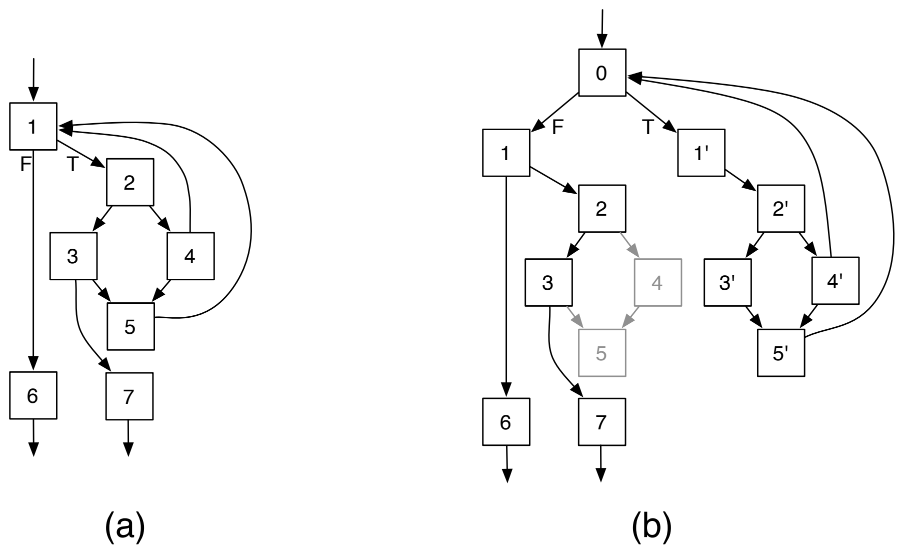


Figure 8: (a) A loop in CFG with two back edges and two exits. (b) The CFG of the transformed loop.

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).

After the transformation, all loops in the program become while loops and our method applies. Since the new generated loop predicate is unknown, to build the loop predicate in the reverse program, we cannot use the first approach proposed above any more.

Because those two newly created branch bodies share the same code in the forward program, any instrumentation will also be shared between them. For example, if a value defined in the false body above needs a state saving, in the forward program we can only perform the state saving in the loop, which results in saving a value many times and then larger time overhead. For this reason, we could forbid state savings on any variable in the false body. Properly defining the cost of this operation may help to get the better search result.

CHAPTER III

SYNTHESIS FOR PROGRAMS WITH ARRAYS

In this chapter, we extend the previous method to handling arrays in both loop-free programs and those with loops. We will still use the same framework. That is, we will build a VSG for the program with arrays, then perform a search on it to build a RG. Then we translate the RG into the source code. To build the VSG for arrays, we apply a modified Array SSA based on [25], and define several special VSG nodes for arrays. To represent the equality between two arrays, we employ the array subregion as the constraint. During the search those subregions will be calculated to guarantee that all array elements will be retrieved. We also develop a demand-driven method to retrieve array elements from a loop.

3.1 Handling loop-free programs with arrays

We first consider how to extend the VSG and RG machinery to handle arrays in loop-free programs. (We treat loops in Section 3.2.) The key idea behind our method is explicit representation of array subregions (subsets of array elements) combined with a modified form of Array SSA [25].

3.1.1 Array subregion representations and operations

Given an array a with length $a.length$, all elements can be represented by their indices as an interval $[0, a.length)$. We denote this index set, $[0, a.length)$, as the *universal set*, $\mathcal{U}(a)$. A (strict) subregion of a is a (strict) integer subset of the universal set.

Among all possible subregions of an array, notable cases we will consider include:

- The subregion containing a single index i, denoted $\{i\}$, where i is a constant or an SSA name.

- The set of all indices other than i, or $\overline{\{i\}} = \mathcal{U}(a) - \{i\}$.
- The triplet $[p : s : q]$ is the set of all indices starting from p up to (and possibly including) q with stride s. We use the shorthand $[p : q]$ when $s = 1$.

We need index set operations so that our analysis algorithm can conclude whether or not we have restored all elements of the array on all paths. However, our analysis, being symbolic at compile-time, will also need to be conservative. For example, given an intersection $\{i\} \cap \{j\}$, the result of it could be $\{i\}$ or $\emptyset$, depending on whether $i = j$ or $i \neq j$, which may be indeterminate at compile-time. As such, key operations we will use are:

$$\{i\} \cap \{j\} = \begin{cases} \{i\} & \text{if } i = j \\ \emptyset & \text{if } i \neq j \end{cases}$$

$$\{i\} \cap \overline{\{j\}} = \begin{cases} \{i\} & \text{if } i \neq j \\ \emptyset & \text{if } i = j \end{cases}$$

$$\{i\} \cup \{j\} = \begin{cases} \{i\} & \text{if } i = j \\ \{i\} \cup \{j\} & \text{if } i \neq j \end{cases}$$

$$\{i\} \cup \overline{\{j\}} = \begin{cases} \mathcal{U} & \text{if } i = j \\ \overline{\{j\}} & \text{if } i \neq j \end{cases}$$

$$\{i\} \cap [p : q] = \begin{cases} [p : q] & \text{if } i \geq p \text{ and } i \leq q \\ \emptyset & \text{if } i < p \text{ or } i > q \end{cases}$$

Because the VSG reveals equalities between values, we can use it to check if $i = j$ by starting a path search from i to j. To check inequality, we can use the previously proposed *inequality graph* [7]. In this representation, each node is a constant or an SSA name; each directed edge $x \xrightarrow{c} y$ represents $x - y \leq c$. The inequalities are obtained from, for instance, branch predicates like `if`$(x > y)$ and assignments like

$x = y + c$, where c is a constant. From the inequality graph, checking whether $x \neq y$ is equal to checking if $x - y \leq -1$ or $y - x \leq -1$.

As our analysis manipulates and simplifies set operations, we may need to normalize the operations according to the set operation laws, including identity laws, domination laws, idempotentency, commutativity, associativity, distributivity, among others.

3.1.2 Modified δ function in Array SSA

In scalar SSA, the entire array receives a new version number even when just a single element is modified, i.e., even assigning one element effectively kills all preceding array definitions. To better support array-based programs, we adapt Array SSA [25]. In Array SSA, defining an array element only kills the previous definition of that element instead of the whole array. Therefore, it more accurately represent the use-def relations between array subregions.

Our modified form of Array SSA is simple: after an array element is modified, we (a) assign a new version to the corresponding array, and also (b) define a δ function (as in Array SSA) to maintain equality relations between the unmodified array subregions in the new array and the previous array. Because we only care equality relations instead of def-use, our modified Array SSA has fewer SSA names and simpler δ functions compared to the original one. For example, consider the following program.

$$\begin{aligned}&\mathit{int}\ a[N];\\&a[i] := 0;\\&a[j] := a[j] + 1;\end{aligned}$$

The program in our modified array SSA is shown below:

$$int\ a_0[N];$$

$$a_1[i] := 0;$$

$$[a_1, \overline{\{i\}}] := \delta([a_0, \overline{\{i\}}]);$$

$$a_2[j] := a_1[j] + 1;$$

$$[a_2, \overline{\{j\}}] := \delta([a_1, \overline{\{j\}}]);$$

Note that when $a[i]$ is modified, we give the array a a new version 1 as in original SSA, and just after this definition, we create a δ function $[a_1, \overline{\{i\}}] := \delta([a_0, \overline{\{i\}}])$ that defines the subregion $\overline{\{i\}}$ of a_1 by the same subregion of a_0. From this δ function, we know a_0 and a_1 have identical elements in the subregion $\overline{\{i\}}$. We use the notation $(a_0 \equiv a_1)@\overline{\{i\}}$ to represent such a relationship; thus, in this example, we also have $(a_1 \equiv a_2)@\overline{\{j\}}$ from the other δ function, $[a_2, \overline{\{j\}}] := \delta([a_1, \overline{\{j\}}])$.

In addition, the ϕ functions that appear in SSA, when defined for arrays with several array definitions from different control flow paths as the arguments, have the same meaning as those for scalars. That is, for a ϕ function $a_1 := \phi(a_2, a_3)$, we have $(a_1 \equiv a_2)@\mathcal{U}(a)$ and $(a_1 \equiv a_3)@\mathcal{U}(a)$ with different control flow path sets as conditions.

3.1.3 Arrays in the VSG

Since an array is a collection of values, we would like the VSG to be able to express equalities between individual values where needed. Here, we describe a technique for doing so.

Let a_u be version u of an array definition. We augment the VSG with an *array node* to represent it, and refer to this array node by a_u directly. Any element $a_u[i]$ is a scalar value and may still have a scalar value node in the VSG. A δ relation, $[a_u, \overline{\{i\}}] = \delta([a_v, \overline{\{i\}}])$, expresses the equalities $a_u[j] = a_v[j], \forall j \in \overline{\{i\}}$. To represent this relation, we add an *array edge* in the VSG between the array nodes a_u and a_v, and attach the subregion $\overline{\{i\}}$ to this array edge. For each array access $a_u[i]$ in the program, we add a relation between this element and the array a_u by adding an edge connecting the corresponding two nodes in the VSG. We call this edge as an *array access edge*. Similarly, we attach the subregion $\{i\}$ to this edge. As before, every edge in the VSG is also attached with a control flow path set, including array and array access edges.

Figure 9 shows the VSG built for the above array example above. Each array node is a square, to differentiate from circular nodes for scalars. Since there is only one control flow path in this program, the path information on each VSG edge is not shown here.

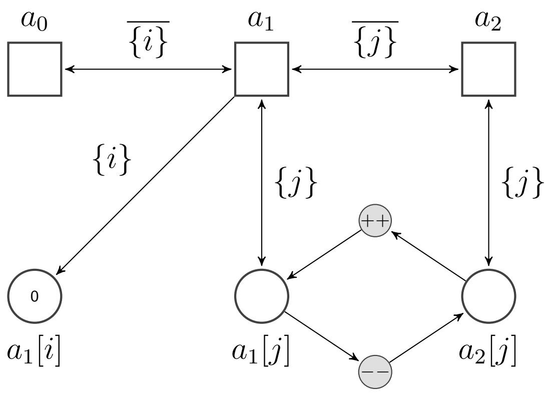


Figure 9: The VSG.

For each ϕ function defined for arrays, in the VSG we build a ϕ array node and connect it to all of its arguments. Again, we attach a control flow path set and the full array region to the edge.

3.1.4 State saving on an array and its elements

Recall that in the forward program we may choose to save state; to enable this possibility, we create a state saving node in the VSG and connect all value nodes to it. During the search, selecting such a state saving edge generates a state saving statement in the forward program. If we wish to regard state saving as expensive, we can attaching costs to all edges and penalize state saving by assigning higher weights to state saving edges. It is possible to formulate the search algorithm to account for such costs [17].

For each array node a_u in the VSG, we also connect it to the state saving node using an edge that we call a *state saving array edge*. The subregion on this edge is the full region $\mathcal{U}(a)$. For example, Figure 10 shows the VSG with a new added state saving node and three state saving array edges for the VSG shown in Figure 9. During the search, the subregion on a state saving array edge will be updated, and the cost of this edge is calculated based on the size of the updated subregion. Assume the cost of storing an array element is c, and the size of the subregion on the state saving edge in the search result is s, then the cost of this edge is $s \times c$.

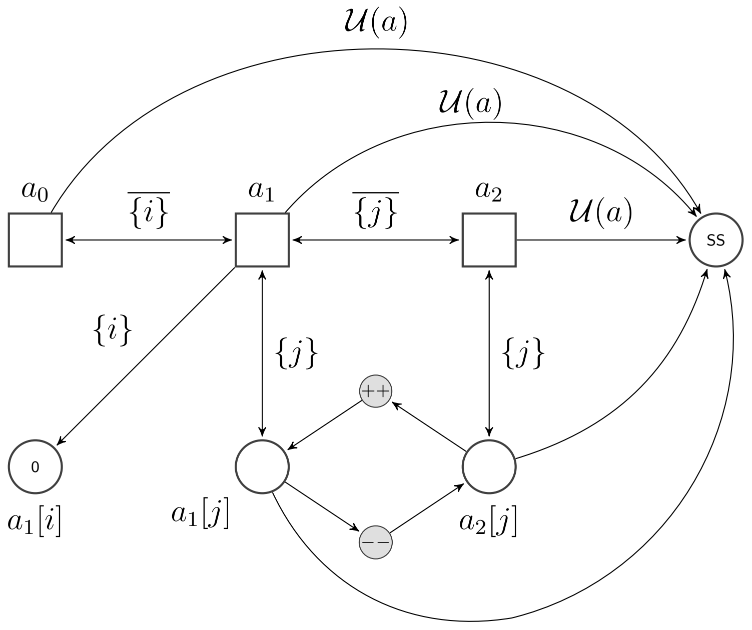


Figure 10: A state saving node connecting all value nodes.

3.1.5 Search the VSG to retrieve an array

For array programs, we need to modify the scalar VSG search procedure of Section **??** to take the appropriate action when it encounters an array node.

There are three scenarios in which a search may reach an array node: (a) at the start of the search, when the whole array a_0 needs to be retrieved; (b) when the search reaches the array node a_u from an element node $a_u[i]$, while searching for the subregion $\{i\}$; or (c) when the search reaches the array node a_u from another array node a_v. In any of these cases, there will be a particular subregion that is the search target. The search needs to explore the incident edges in order to find all values of the array elements in this target subregion. That is, let R_t be the target subregion whose values we seek at some point during the search. Suppose the search selects a particular outgoing edge e with subregion R_e. Then, $R_e \cap R^t$ is the subregion of the array that could be retrieved using this edge. The search may need to continue to select edges until their union $\bigcup R_i = R^t$.

Before giving a search algorithm for array-based programs, we first state the desired properties of the search result, i.e., the RG. Similar properties of a RG for scalar programs appeared in the original work we are using [17]. Here, we generalize these properties for both scalar and array value nodes. To formalize these properties, let $\mathcal{G}$ be a RG and consider a filtered RG, $\mathcal{G}_p$, under a control flow path p. That is, $\mathcal{G}_p$ is a graph obtained from $\mathcal{G}$ by selecting only edges with and their incident nodes if the control flow path set the edge contains p. The formal properties appear in Table 1 and apply to $\mathcal{G}_p, \forall p$. Here, we summarize the key intuition behind each property:

- Property I states that to retrieve a whole array is to retrieve all elements thereof.
- Property II states that every desired array element during the search must be retrieved.
- Property III states that the value of each array element needs to be retrieved only once.
- Property IV forbids cyclic data dependence in the RG: given a loop-free program, we wish to build loop-free forward and reverse programs, which should

Table 1: The properties of the RG. The "scalar" column shows the properties as stated in other work [17]; the "array" column shows our generalizations for array-based programs.

	Scalar node	Array node
I	For each target scalar node n, if it is not an available node, then $OutDegree(n) > 0$.	For each target array node n, if it is not an available node, then $OutDegree(n) > 0$, and $\bigcup_{out \in OutEdges(n)} = \mathcal{U}(a)$.
II	For each scalar node n that is neither a target node nor an available node, then if $InDegree(n) > 0$, then $OutDegree(n) > 0$.	For each array node n that is neither a target node nor an available node, then if $InDegree(n) > 0$, then $OutDegree(n) > 0$, and $\bigcup_{out \in \text{OutEdges}(n)} R_{out} = \bigcup_{in \in \text{InEdges}(n)} R_{in}$.
III	For each scalar node n, $OutDegree(n) \leq 1$.	For each array node n, if $OutDegree(n) > 1$, then for $e, f \in OutEdges(n)$, $e \neq f$,$R_e \cap R_f = \emptyset$.
IV	There is no directed cycle that contains no array node.	For each directed cycle with array and array access edges $e_1 \dots e_n$, $\bigcap_{i=1}^{n} R_{e_i} = \emptyset$.

not have any cyclic data dependences.

The formal search algorithm for array nodes appears in Algorithm 5. We retrieve each desired array a_0 by starting a search `SearchSubregion`$(a_0, \mathcal{U}(a), \emptyset)$, thereby fulfilling Property I. In Lines 7-13, the algorithm tries to retrieve a subset of `subregion` through each outgoing edge from `target`, and those search results are sorted in Line 14 by cost. Lines 15-21 pick the best search results while also satisfying Property III. Note that in Line 20, if a candidate edge in `route` already exists in `resultRoute`, we update the subregion on it in `resultRoute` to be the union of itself and the subregion on the same edge in `route`. Line 22 checks whether Property II is satisfied. Because of the existence of state saving edges, Property II can always be eventually satisfied. Property IV is satisfied by the cycle checks in Lines 12 & 18. Finally, the search result is returned.

This algorithm works only on array nodes. When the search reaches a scalar value node, we invoke the earlier version of this algorithm for scalars [17], which is similar to Algorithm 5 but without the operations related to subregions. In addition, note that Algorithm 5 is run for each control flow path. We need to perform this algorithm on all control flow paths in the original program to retrieve any desired value.

This last fact reveals a weakness of the scheme, which was already a weakness of the scalar case and related path profiling algorithms: the asymptotic cost of search grows with the number of control flow paths. This cost is exponential in the program size in the worst case. However, for the vast majority of reasonably structured programs, the absolute number of such paths per subprogram is not typically very large, thereby yielding reasonable compile-time costs.

An additional detail is that we must also retrieve all indices that appear in an array element node and subregions on the edges. To do so, we start another search to retrieve those index values and then combine the search result to the RG.

3.1.6 Generating code from Route Graph

Recall that a RG shows explicit data dependences in the corresponding reverse program, and that we need to translate the edges in the RG, visited in the reverse topological order, into statements in the reverse program. Our scheme gives a concrete translation algorithm as shown in Algorithm 4. The biggest twist in the array case is the presence of subregions on array edges (including access and state saving edges). Array edges and array access edges will not be translated into any statements because they don't really define any values—array edges come from δ functions which are pseudo-definitions that will not appear in reverse program. Therefore, the subregion on such an edge does not affect the generated code, as long as it is not an empty set. If the subregion on an edge is an empty set, this edge should be removed from

Algorithm 5: Search for a subregion of an array in the VSG.

Input: The search start point target which is an array node, the target subregion subregion, and the already collected edges in the previous search currentRoute.

```
1  SearchSubregion(target, subregion, currentRoute)
2  begin
3      resultRoute $\leftarrow \emptyset$, subRoutes $\leftarrow \emptyset$, r $\leftarrow \emptyset$
4      if target is available then
5          Add target to resultRoute
6          return resultRoute
7      foreach edge $\in$ OutEdges(target) do
8          edge.subregion $\leftarrow$ edge.subregion $\cap$ subregion
9          if edge.subregion $=$ $\emptyset$ then continue
10         newRoute $\leftarrow$ SearchSubRoute(edge.target, newSubregion,
           currentRoute $+$ edge)
11         Add edge to newRoute
12         if HasNoCycle( currentRoute $+$ newRoute) then
13             Add newRoute to subRoutes
14     Sort subRoutes according to cost in ascending order
15     foreach route in subRoutes do
16         route.subregion $\leftarrow$ route.subregion $-$ r
17         if route.subregion $= \emptyset$ then continue
18         if HasNoCycle( resultRoute $+$ route) then
19             r $\leftarrow$ r $\cup$ route.subregion
20             Add route to resultRoute
21             if r $=$ subregion then break
22     if r $\neq$ subregion then return $\emptyset$
23     Add target to resultRoute
24     return resultRoute
```

the RG. Since a subregion is symbolically represented, as we discussed before it may be an empty set conditionally. For example, given a subregion $\{i\} \cap \{j\}$ where we cannot determine if $i \neq j$ at compile time, we also cannot determine if it is an empty set or not. If we still keep this edge during code generation, and at runtime $i \neq j$, then we may retrieve some additional values that are unnecessary to be retrieved.

To avoid such redundant retrievals, in the reverse program we can add a condition as a runtime check to all code translated from this edge and following edges. This extra condition guarantees the subregion cannot be empty. For example, the condition to be added for the subregion $\{i\} \cap \{j\}$ is $if\ (i \neq j)$.

An alternative method is to avoid adding runtime conditions, which makes the code generation easier and reduces the number of conditions in the reverse program. The price is that we may recover some values which are not really needed, or we may retrieve a value more than once. However, such redundant restores do not affect the correctness of the reverse program.

Let us now consider the overall scheme for the running example used in this section. Assume that the input is a_0, that the output is a_2, and that the indices i and j are available values. To build a reverse program with input a_2 and output a_0 from the VSG shown in Figure 10, we will search the for values of all elements of a_0 from a_2. The search result is shown in Figure 11.

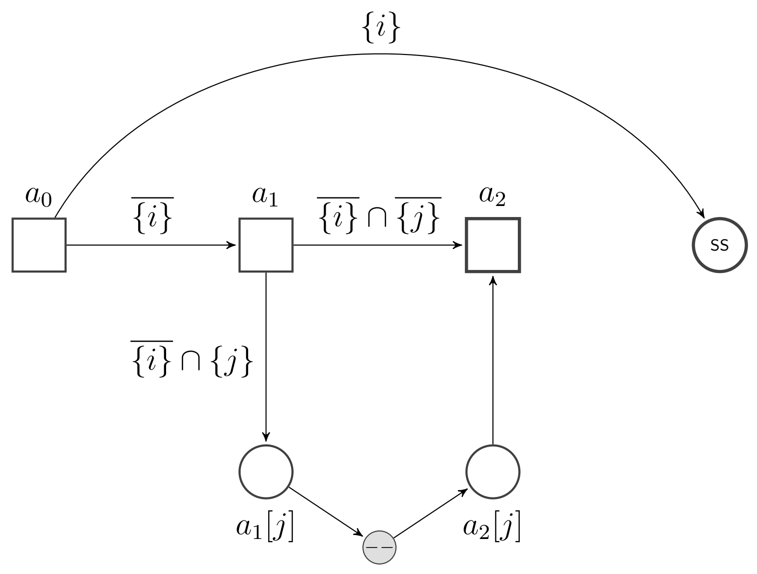

$\{i\}$
a_0
$\overline{\{i\}}$
a_1
$\overline{\{i\}} \cap \overline{\{j\}}$
a_2
ss
$\overline{\{i\}} \cap \{j\}$
$a_1[j]$
$a_2[j]$
--

Figure 11: The RG built from the search on the VSG shown in Figure 10.

From this RG, the overall algorithm will generate the following forward (first) and the reverse (second) programs:

```
store(a[i]);
a[i] = 0;
a[j] = a[j] + 1;
```

```
if (i != j)
    a[j] = a[j] - 1;
restore(a[i]);
```

Observe that the condition $if(i \neq j)$ in the reverse program is generated according to the subregion $\overline{\{i\}} \cap \{j\}$ on the edge $a_1 \rightarrow a_i[j]$, which is necessary to ensure it is not empty. If we remove this condition from the reverse program, we still get a correct result. Moreover, in the case of $i = j$, the operation $a[j] = a[j] - 1$ is unnecessary but it does not have any correctness side effects.

3.2 Handling programs with loops and arrays

In this section we discuss how to retrieve an array or a subregion of it from a do-while loop. (Remember that in Section 2.2.2 a method is proposed to transform an arbitrary loop into a while loop, which can also be further transformed into a do-while loop.) We will consider two scenarios: the array is modified in the loop, and it is just used in the loop.

3.2.1 The array being retrieved is modified in the loop

Given a do-while loop l that modifies an array a, in SSA there are at least four important definitions of a as shown below: the input/output of the loop a_{init} and a_{final}, and the input/output of each iteration a_{in} and a_{out}. Now we consider the problem how to synthesize a loop in the reverse program to retrieve a_{init}.

```
$/*\ Input : a_{init}\ */$
$do\ \{$
    $a_{in} := \mu(a_{init}, a_{out});$
    $...;$
    $a_{out} := ...;$
$\}\ while\ (...);$
$a_{final} := \eta(a_{out});$
```

A simple idea to retrieve a_{init} is using the method developed for scalars [16]: in each iteration, we retrieve all elements of a_{in} from a_{out} and/or possibly other values. We don't consider to store values in a loop in the forward program, because it always brings high cost. Consequently, in each iteration it is not guaranteed that all elements of a_{in} can be retrieved, but possibly only a subregion of it. However, the retrieval of a subregion in an iteration may require another subregion of the same or another array in the next iteration, and it difficult to check the equalities/inequalities between values defined in different iterations, in order to resolve the set operations during the search. In addition, because of the data dependences between two successive iterations, the generated loop will have the opposite iteration order to the original one. Given a variable v, let v^t be its value in the iteration t. Assume this value is needed in the generated loop. Then we say the generated loop has the the identical or opposite iteration order to the original loop, if this variable has the same value as v^t in the iteration t or $l.len-t-1$ of the generated loop respectively, where $l.len$ is the number of iterations for both original and generated loop. However, not like scalars, there may not exist any data dependences between two iterations in an array that is updated in a loop, and then we may be able to choose either identical or opposite iteration order for the generated loop. In some cases we may need this flexibility (we will see such a case in Section 3.2.3).

To overcome those two drawbacks, we develop a new searching strategy, which is based on the fact that each element can be retrieved through the loop only if it is used inside. Indeed, if that element is not used in the loop, in the VSG we don't have any equality information of it. Assume the array a is used in the loop through the index i. We will try to retrieve the value of $a_{init}[i^t]$ through $a_{in}[i^t]$ if $a_{init}[i^t] = a_{in}[i^t], \forall t \in [0, l.len)$. The successful retrieval of $a_{init}[i^t], \forall t \in [0, l.len)$ then leads to the retrieval of the subregion $\bigcup_{t\in[0,l.len)} \{i^t\}$ of a_{init}. Below we will introduce how we build the VSG as shown in Figure 12 to enable this approach and how to search the value of $a_{init}[i]$ in each iteration.

3.2.1.1 Build the VSG for arrays modified in a loop

In SSA, the input of each iteration a_{in} is defined by a μ function, which is a special ϕ function whose arguments contain definitions from both inside and outside of the loop. Given such a μ function: $a_{in} = \mu(a_{init}, a_{out})$, let's consider at the beginning of the iteration t, what is the subregion $\mathcal{R}_I^t$ such that $(a_{init} \equiv a_{in})@\mathcal{R}_I^t$.

Let $R_{def}^a(m, n)$ define all indices at which the array a is modified from the beginning of the iteration m to the beginning of the iteration n. Also, let $\mathcal{I}_{use}^a$ and $\mathcal{I}_{def}^a$ be two sets containing all indices of a from which the elements of a are used and defined in the loop respectively. Since each modification to a is made through an index in $\mathcal{I}_{def}^a$, we have:

$$R_{def}^a(m, n) = \bigcup_{i \in \mathcal{I}_{def}^a} \bigcup_{t \in [m,n)} \{i^t\}$$

And $R_{def}^a(m, m) = \emptyset$. Then $\mathcal{R}_I^t$ is a complementary set of $R_{def}^a(0, t)$:

$$\mathcal{R}_I^t = \overline{R_{def}^a(0, t)}$$

To show $(a_{init} \equiv a_{in})@\mathcal{R}_I^t$, we attach $\mathcal{R}_I^t$ to the VSG edge $a_{init} \leftrightarrow a_{in}$ as shown in Figure 12. With the assist of $\mathcal{R}_I^t$, checking if $a_{init}[i^t] = a_{in}[i^t]$ becomes checking if $i^t \in \mathcal{R}_I^t$.

Based on the μ function $a_{in} = \mu(a_{init}, a_{out})$, we also connect a_{in} and a_{out} with two directed edges. However, each edge shows an equality across iterations. The edge $a_{in} \rightarrow a_{out}$ implies the data dependence from values in the iteration t to the values in the iteration $t - 1$. If this edge is selected in the RG, the generated loop must follow the same iteration order as the original loop. We call this edge a *forward edge* as in [16]. Similarly, if the edge $a_{out} \rightarrow a_{in}$ is selected in the RG, the generated loop should have the opposite iteration order to the original loop, and we call this edge a *reverse edge*. Apparently, the search result of one value (either scalar or array) cannot contain both forward and reverse edges, since the generated loop can only have one iteration order. To show this difference, in Figure 12, a forward edge is shown as a dotted edge, and a reverse edge is shown as a dashed edge.

The output of the loop a_{final} is the output of the last iteration. At the end of the iteration t, let $\mathcal{R}_O^t$ be a subregion in which the elements of a_{out} will not be modified in the following iterations and hence $(a_{out} \equiv a_{final})@\mathcal{R}_O^t$. Then we have

$$\mathcal{R}_O^t = \overline{R_{def}^a(t+1, l.len)}$$

We add this subregion to the edge $a_{out} \leftrightarrow a_{final}$. With the help of $\mathcal{R}_O^t$, during the search if $a_{out}[i^t]$ is required and $i^t \in \mathcal{R}_O^t$, the search can exit the loop through the edge $a_{out} \rightarrow a_{final}$. Otherwise, the search will pick the reverse edge $a_{out} \rightarrow a_{in}$ and enter the next iteration.

Finally, we add a summary edge between a_{init} and a_{final} with the subregion $\mathcal{R}_U = \overline{R_{def}^a(0, l.len)}$, in which all elements remain unchanged after the loop. Theoretically, each element that is not modified in the loop can be retrieved through this edge.

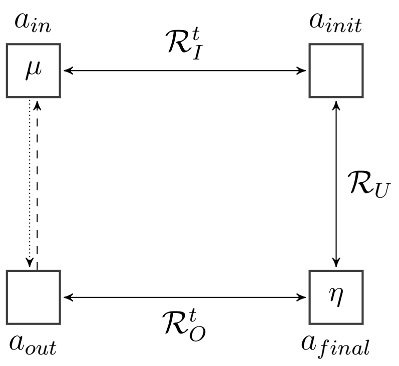


Figure 12: The relations between four definitions of an array in a loop. The dashed edge (reverse edge) implies the data dependence from an iteration to the next one. The dotted edge (forward edge) implies the data dependence from an iteration to the prior one.

3.2.1.2 The search algorithm

Now we present an algorithm of retrieving a_{init} in the VSG as shown in Figure 12. We denote $\texttt{Search}(a_{init}, \{i^t\})$ as a search on the node a_{init} for the index i^t, where $i \in \mathcal{I}_{use}^a$. Also let $\mathcal{A}_{out}$ be a set of all array definitions as the output of the iteration. We assume there is no nested loops, which we will discuss later.

For each $i \in \mathcal{I}_{use}^a$, we do $\texttt{Search}(a_{init}, \{i^t\})$. If it is successful, the retrieved

Algorithm 6: The algorithm of `Search`$(a_{init}, \{i^t\})$

1. If $i^t \in \mathcal{R}_U$, $a_{init}[i^t]$ can be retrieved through $a_{init} \xrightarrow{\mathcal{R}_U} a_{final}$.

2. Else, if $i^t \in \mathcal{R}_I^t$, the search continues through the edge $a_{init} \xrightarrow{\mathcal{R}_I^t} a_{in}$ and becomes `Search`$(a_{in}, \{i^t\})$, which is the same as the search for loop-free programs (the loop body is treated as a loop-free program), until the following situations occur.

   (a) If the search reaches an array node b_{out} with subregion $\mathcal{S}$, where $b_{out} \in \mathcal{A}_{out}$, then

      i. If $\mathcal{S} \subseteq \mathcal{R}_O^t$, the search exits the loop through the edge $b_{out} \xrightarrow{\mathcal{R}_O^t} b_{final}$, and becomes `Search`$(b_{out}, \mathcal{S})$.

      ii. Else, the search continues through the edge $b_{out} \longrightarrow b_{in}$ and enters the next iteration (and hence t is incremented by one) and becomes `Search`$(b_{in}, \mathcal{S})$.

   (b) If the search reaches a scalar value node s_{out}, which is an output of the iteration, then it continues through the edge $s_{out} \rightarrow s_{in}$ and enters the next iteration. Note that the search will then apply the method proposed to reverse programs with loops for scalar values [16].

   (c) If the search reaches a value node defined outside of the loop, then it continues using the method for loop-free programs.

subregion of a_{init} is $\bigcup_{t\in[0,l.len)}\{i^t\}$. The final retrieved subregion is the union of each retrieved subregion for each index; the elements not in this subregion still need to be retrieved outside of the loop. If the search result does not contain reverse edges, the generated loop can have arbitrary iteration order (although we only consider the same and opposite order as the original loop). If the search reaches the next iteration in Step 2-(a)-ii as $\texttt{Search}(b_{in}^{t+1}, \mathcal{S})$, and it is quite possible that $\mathcal{S}$ contains values defined in the iteration t. However, it is difficult to check the equality/inequality between two values in two different iterations as required during the search. In Section 3.2.2, we will propose a solution to this issue.

Once $\texttt{Search}(a_{init}, \{i^t\})$ is successful, we also need to retrieve i^t, and the search result should obey the same iteration order as the generated loop.

3.2.1.3 Induction variables as indices

In the search algorithm above, checking the set membership between an index and a subregion is essential. This is difficult for an arbitrary index and subregion, but if all indices in $\mathcal{I}_{use}^a$ and $\mathcal{I}_{def}^a$ are induction variables, their properties can make it possible to accomplish the membership inspection at compile time.

An *induction variable* [29] is a variable whose value is systemically incremented or decremented by a constant value in a loop. Given an induction variable i with an initial value i_{init} and the step s_i. Let i_{in}^t and i_{out}^t be the input and output value of the iteration t, then we have $i_{in}^t = i_{init} + s_i \times t$ and $i_{out}^t = i_{init} + s_i \times (t+1)$ respectively. Also denote i's output value of the loop by i_{final}, and $i_{final} = i_{init} + s_i \times l.len$. Then we have

$$\bigcup_{t\in[0,l.len)} i_{in}^t = [i_{init} : s_i : i_{final} - s_i]$$

$$\bigcup_{t\in[0,l.len)} i_{out}^t = [i_{init} + s_i : s_i : i_{final}]$$

.

This triplet representation not only makes it easier to represent the result of set operations like union on several such subregions, but also makes it possible to check the set membership at compile time. For example, in the search algorithm we need to know if $i^t \in \mathcal{R}_I^t$. If all indices in $\mathcal{I}_{def}^a$ are induction variables, then we have

$$\mathcal{R}_I^t = \overline{\bigcup_{j \in \mathcal{I}_{def}^a} [j^0 : s_j : j^{t-1}]} = \overline{\bigcup_{j \in \mathcal{I}_{def}^a} [j^0 : s_j : j^t - s_j]}$$

Then $i^t \in \mathcal{R}_I^t \Leftrightarrow i^t \notin [j^0 : s_j : j^t - s_j], \forall j \in \mathcal{I}_{def}^a$. When $s_j = 1$, we have $i^t \notin [j^0 : j^t - s_j] \Leftrightarrow i^t < j^0 \vee i^t \geq j^t$, which could be resolved by checking the corresponding inequalities. GCD test and Banerjee test can also be used to check the membership. Due to the space limit, we don't discuss those methods here.

3.2.2 The array being retrieved is not modified in the loop

If the array is not modified in the loop, we can also retrieve it through the uses of its elements. Suppose the array to be retrieved is a_0, and it is used through the index i which is a loop variant, then we search the value of $a_0[i]$ in each iteration. The search algorithm is similar to Algorithm 6. The difference is the search will directly enter the Step 2 without the membership inspection, and continue with the same three possible results 2-(a),2-(b), and 2-(c).

There is a special case that retrieving an array element requires the values of other elements in the same array. In this case, the search coming from the array node may reach the array itself and form a cycle. For example, given a loop with loop condition `while`($a_0[i]$`==`$a_0[j]$), in the VSG shown in Figure 13, retrieving $a_0[i^t]$ needs the value of $a_0[j^t]$, where i and j are both loop variants. If there is no other way to retrieve $a_0[j^t]$, the only way to get its value is through $a_0[i^s]$ if $i^s = j^t$, and $s \neq t$. If for every j^t there is a i^s such that $i^s = j^t$, and assume $s < t$, then the search for $a_0[i^t]$ will follow the path $a_0[i^t] \rightarrow a_0[j^t] \rightarrow a_0[i^s] \rightarrow a_0[j^s] \rightarrow a_0[i^r] \rightarrow a_0[j^r] \rightarrow ..., (t > s > r)$ until the iteration number becomes less that 0. Note that the last value in this path should still be recovered in other ways. This strategy works well when i and j are induction variables with the same step, and can also be applied in Algorithm 6 at Step (a)-ii (note that $\mathcal{S}$ is calculated when searching for one element, so that $|\mathcal{S}| \leq 1$, and usually $\mathcal{S}$ only contains an index).

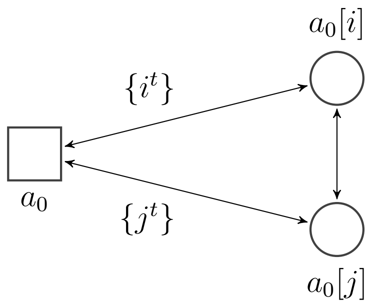
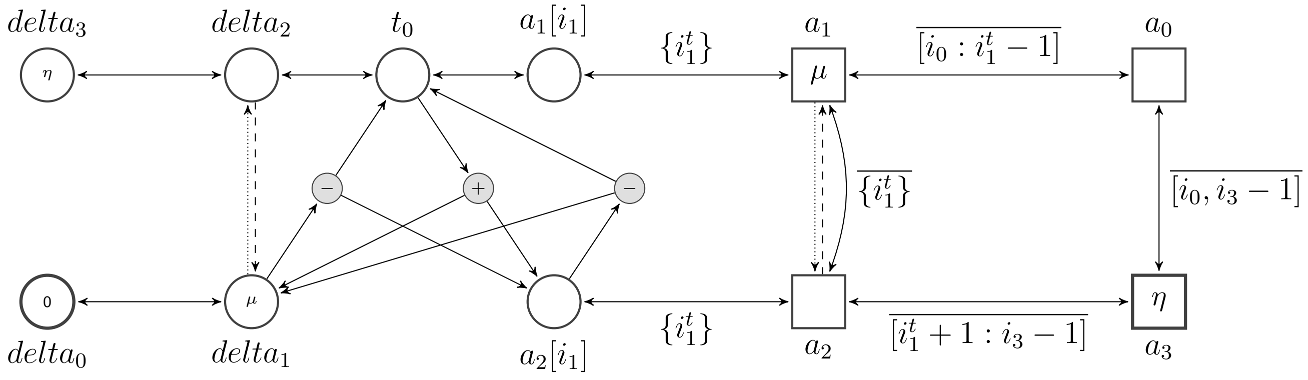


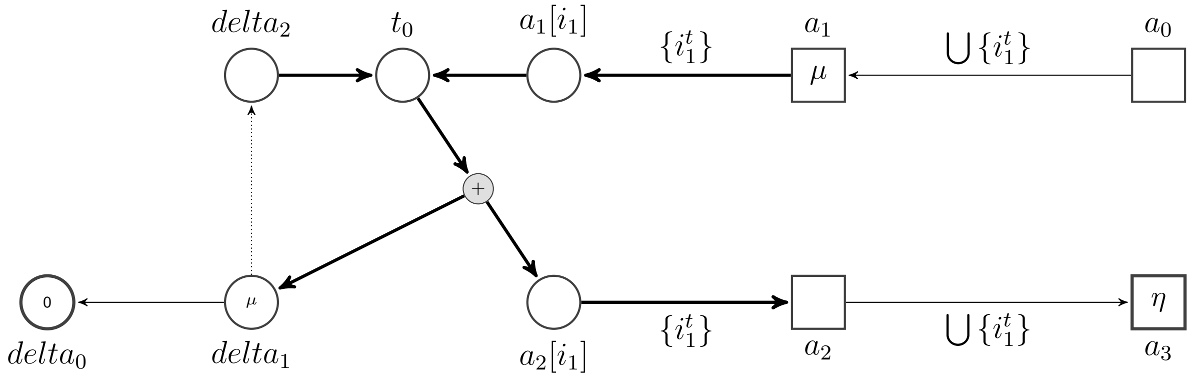
Figure 13: The VSG of a loop with loop condition `while`($a_0[i]$`==`$a_0[j]$).

3.2.3 A case study

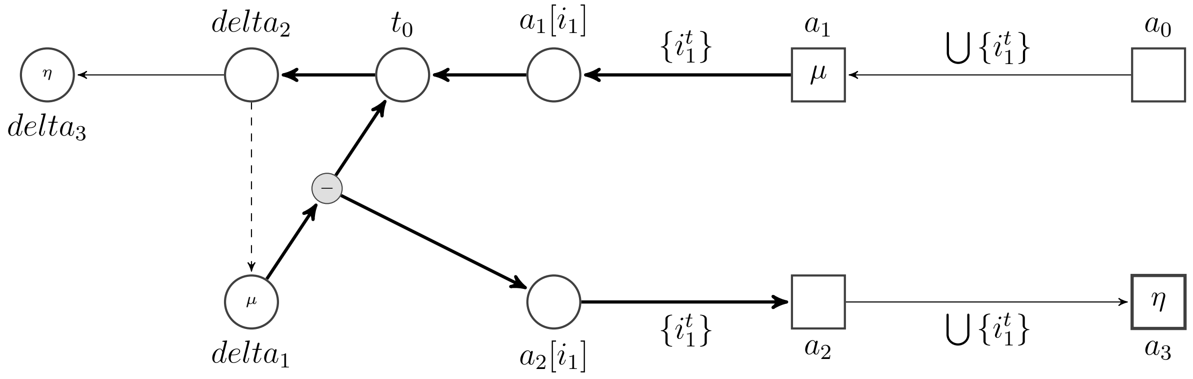
As an example, let's look at how to reverse a delta encoding program, which is shown below together with its SSA form. The VSG is built and shown in Figure 14(a)&15(a). There are three available value nodes: a_3, which is the output of the program; $delta_0$, which is a constant value; and i_3, whose value N is calculated by solving constraints [16]. Their corresponding value nodes are shown in bold in the VSG. Our goal is searching the VSG for a_0.



(a) The VSG.

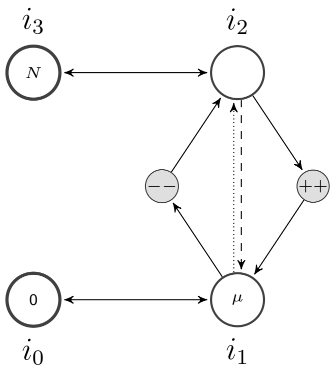


(b) The RG for the first search result.

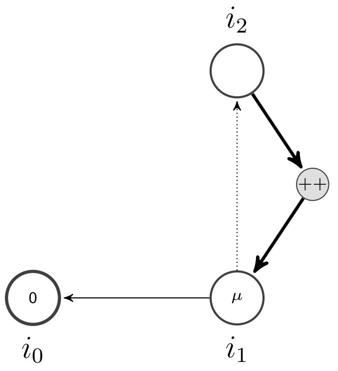


(c) The RG for the second search result.

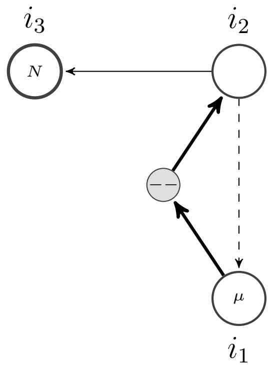
Figure 14: The VSGs and RGs (Part 1).



(a) The VSG of i.



(b) The RG of i for the first search result.



(c) The RG of i for the second search result.

Figure 15: The VSGs and RGs (Part 2).

Original Program	SSA Form
	$/*\ \mathit{Input}: a_0\ */$
	$\mathit{int}\ \mathit{delta}_0 := 0,\ i_0 := 0;$
	$\mathit{do}\ \{$
$/*\ \mathit{Input}: a\ */$	$i_1 := \mu(i_0, i_3);$
$\mathit{int}\ \mathit{delta} := 0;$	$a_1 := \mu(a_0, a_2);$
$\mathit{int}\ i := 0;$	$\mathit{delta}_1 := \mu(\mathit{delta}_0, \mathit{delta}_2);$
$\mathit{do}\ \{$	$\mathit{int}\ t_0 := a_1[i_1];$
$\mathit{int}\ t := a[i];$	$a_2[i_1] := t_0 - \mathit{delta}_1;$
$a[i] := t - \mathit{delta};$	$[a_2, \overline{\{i_1\}}] := \delta([a_1, \overline{\{i_1\}}]);$
$\mathit{delta} := t;$	$\mathit{delta}_2 := t_0;$
$i := i + 1;$	$i_2 := i_1 + 1;$
$\}\ \mathit{while}(i < N);$	$\}\ \mathit{while}\ (i_2 < N);$
$/*\ \mathit{Output}: a\ */$	$i_3 := \eta(i_2);$
	$a_3 := \eta(a_2);$
	$\mathit{delta}_3 := \eta(\mathit{delta}_2);$
	$/*\ \mathit{Output}: a_3\ */$

There is only one use of the element of a_1 in the loop through the index i_1. Therefore, we inquire the value of $a_1[i_1]$ for all iterations. At the iteration t, from the subregion on the edge $a_0 \leftrightarrow a_1$ as shown in Figure 14(a), we have $(a_0 \equiv a_1)@\overline{[i_0 : i_1^t - 1]}$. Because $i_1^t > i_1^t - 1$, and hence $i_1^t \notin \overline{[i_0 : i_1^t - 1]}$, then we have $a_0[i_1^t] = a_1[i_1^t]$. On the VSG, we can get $a_1[i_1^t]$ through t_0 from $a_2[i_1^t] + delta_1^t$, and then the search forks to two directions: one begins with $a_2[i_1^t]$ and one begins with $delta_1^t$.

The search for $a_2[i_1]$ then passes through the edge $a_2[i_1^t] \rightarrow a_2$ and reaches an output of the iteration a_2. According to the searching rule, we inspect if $a_2[i_1^t] = a_3[i_1^t]$ by checking if i_1^t belongs to the subregion on the edge $a_2 \rightarrow a_3$ which is $\overline{[i_1^t + 1 : i_3 - 1]}$. Because $i_1^t < i_1^t + 1$, the answer is true. The search then exits the loop and ends successfully at the available node a_3.

The search for $delta_1$ follows the searching rule in [16]. Figure 14(b) shows one search result. As the value of i_1 is required as the index of a_1 and a_2, we start another search for it. Note that previously a forward edge $delta_1 \rightarrow delta_2$ is already selected, forcing the generated loop to have the same iteration order as the original loop. Therefore, during the search for i_1 the reverse edge $i_2 \rightarrow i_1$ cannot be selected. The search result of i_1 is shown in Figure 15(b). In the reverse program we build the loop body from the bold edges in Figure 14(b)&15(b). The loop condition of the generated loop is built as the same one in the original loop. The synthesized reverse program is shown below. Since there is no state saving used, the forward program is identical to the original one.

```
int delta = 0;
int i = 0;
do {
    int t = a[i] + delta;
    a[i] = t;
    delta = t;
    i = i + 1;
} while (i < N);
```

Figure 14(c)&15(c) show another search result that contains reverse edges. However, since the value $delta_3$ is unknown, it needs to be stored in the forward program (there should be an edge connecting $delta_3$ to the state saving node, which is omitted here). The resulted reverse program is shown below. We prefer the first result because there is no state saving used.

```
int delta;
restore(delta);
int i = N;
do {
    i = i - 1;
    int t = delta;
    delta = t - a[i];
    a[i] = t;
} while (i != 0);
```

3.2.4 Handling nested loops

Handling nested loops is similar to handling a single loop. Assume there is an outer loop l_{out} and an inner loop l_{in}. Given an index i of the array a in the inner loop, let $i^{s,t}$ be the value of i in the iteration s of l_{out} and the iteration t of l_{in}. Then from the view of l_{out}, in the iteration s we try to retrieve the subregion $\bigcup_{t\in[0,l_{in}.len)} \{i^{s,t}\}$ of a. If we can retrieve such a subregion for every $s \in S$, the final subregion of a retrieved is $\bigcup_{s\in S} \bigcup_{t\in[0,l_{in}.len)} \{i^{s,t}\}$.

3.3 Handling linked data structures

A linked data structure contains elements which have one or more “links” to other elements such that each element could reside in separate memory place from others, comparing to arrays whose elements are always stored together. Typical linked data structures include linked list, tree, graph, etc.. Since an element can be linked by several links from other elements, aliasing prevalently exists among linked data structures. But this is similar to the aliasing brought by indices in arrays.

If we treat the address (normally saved in links) of elements in linked data structure as indices in arrays, it is possible to handle those linked data structures using the method we developed for arrays. Let's take the linked list (we will call it as list briefly) as an example. Let $Node$ define a node in the list and its definition is shown below:

```
struct Node {
    int val;
    Node* next;
};
```

Now suppose we have a list and would like to increment the value in each node by one. The code should look like this:

```
void increment(Node* n) {
    while (n != NULL) {
        n->val++;
        n = n->next;
    }
}
```

Then we perform a transformation on this code by taking each field of $Node$ as an array and the address of the node as the index. Here we create two new arrays: *val* and *next*. Whenever we access the field with the same name by an instance of $Node$, we transform it into an access to an array. That is, for $n \rightarrow val$ we get $val[n]$. The transformed code is shown below:

```
void increment(Node* n) {
    while (n != NULL) {
        val[n]++;
```

```
            n = next[n];
        }
}
```

However, unlike the index of an arrays that may have regular access patterns, we usually don't have enough information of the pointers pointing to linked data structures. We may use some high-level semantics like during the traversal of a list, each element only appear at most once (otherwise a cycle will exist in the list which breaks the definition of it).

CHAPTER IV

SYNTHESIS FOR C++ PROGRAMS

In prior chapters, we target programs in no specific languages, as our method is general and can be applied on any imperative language. In this chapter, we introduce some techniques of handling several high-level constructs of C++ programs. We choose C++ for three reasons: first, Backstroke is based on ROSE compiler [21], which is a C++ source-to-source compiler; second, C++ is an object-oriented (OO) language, and our discussion in this chapter can also be extended to other OO languages like Java; third, initially Backstroke is used to generate reverse functions for OPDES events, and our simulation engines (GTNets [22] and ROSS [9]) are writing in C/C++.

4.1 Normalizing C++ programs

Backstroke is a source-to-source translator. It is based on ROSE, which is a C++ source-to-source compiler. The ROSE compiler parses the source code and transform it into an abstract syntax tree (AST) as the intermediate representation (IR). The AST preserves almost all syntax informations on source level of the source code, such that it can reproduce the similar code as the input together with the transformations as desired. But it is not sufficient for some low level analysis, where other IRs like three address code, in which each statement only has one operation with at most three operands, is easier to analyze. As seen in the prior two chapters, our method heavily depends on SSA form. Building the SSA form on source code is challenging, as the source may contain complex statements or expressions.

To amortize those problems, given an input program, we will firstly normalize it into another form without changing the behavior of the original program. We will break some complex expressions or statements into simpler ones, and try to let each statement only have one side effect. Then the data flow analysis including SSA form can be easily generated. In addition, because the forward program is generated by instrumenting state saving and path recording statements in the original program, normalizing the program can make it easier to find those instrumentation locations in the program.

4.1.1 Forcing a specific execution order of several expressions

In C++, there is a concept called *sequence point* that is used to indicate the execution order of several expressions. A sequence point defines any point in a computer program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been performed. Normally, all expressions in a statement are executed before the next statement. But in one statement, several expressions with side effects may exist. In this case, the C++ standard only guarentees the execution order for the following expressions:

- Comma expression: $expr1, expr2$. In this expression, $expr1$ must be executed before $expr2$, and the returned value of $expr2$ is used as the returned value of this comma expression.

- Logical and/or operator: $expr1$ && $expr2$ / $expr1$ || $expr2$. In this expression, $expr1$ must be executed first, and $expr2$ is executed only if the returned value of $expr1$ is $true/false$ for logical and/or operations. This is called *short-circuit evaluation.*

- Conditional operator: $expr1$? $expr2$: $expr3$. In this expression, $expr1$ will be executed first, and if the returned value is $true$, then $expr2$ is executed; otherwise, $expr3$ is executed.

For other expressions with several side effects, their order is undefined. For example, if we have $++a = ++b$, the resulted value of a can be either b or $b+1$. Although there is no syntax error in this expression, it may bring some problems for our data flow analysis. To remove this ambiguity, we transform this kind of expression into several ones and force an execution order. The new expression will be $++a, ++b, a = b$, in which the comma expressions form a specific execution order of all three side effects.

4.1.2 Facilitating instrumentations

A forward program is always generated by instrumenting the original program with state saving and path recording statements. We have to find the proper place to make the instrumentations. However, some C++ constructs make it a little difficult to find such a place.

Take the logical and operator as an example. The evaluation of a logical and operation follows the short-circuit rule: if the operand on the left hand side is evaluated as $false$, then the operand on the right hand side will not be evaluated. Actually, such an operator implicitly creates additional control flow paths to the program, and during path recording we also have to count those paths, especially if there exists side effects in either operand.

Below is an example showing this situation.

```
if (a > 0 && b++ > c)
    /* true body */
else
    /* false body */
```

The CFG of it is shown in Figure 16. During the path recording, we may have to insert a statement on the dashed edge in this CFG, which is a critical edge (an edge which is neither the only edge leaving its source block, nor the only edge entering its destination block). However, at the source level it is very difficult to insert a statement on this edge.

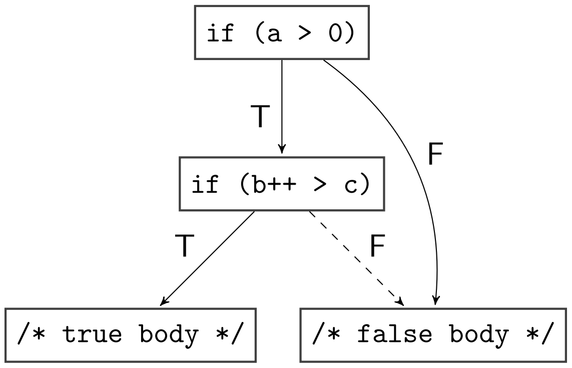


Figure 16: The CFG for a logical and operation `a > 0 && b++ > c` as a predicate.

Our solution is first converting such an operator into a conditional operator, which will be converted into conditional statements later. We convert logical and & or operations into conditional operation according to the rules as below:

$$expr1 \,\&\&\, expr2 \;\rightarrow\; expr1 \;?\; expr2 : false$$

$$expr1 \,||\, expr2 \;\rightarrow\; expr1 \;?\; true : expr2$$

For a conditional operator as a predicate, we will declare a new boolean variable, then assign the result of the conditional operation to it. Consequently, the code above becomes:

```
bool f = a > 0 ? b++ > c : false;
if (f)
    /* true body */
else
    /* false body */
```

We then go ahead convert the conditional operator into a conditional if statement, and separate the statement $f = b++ > c$ into two statements such that each statement only contains one side effect:

```
bool f;
```

```
if (a > 0) {
    f = b > c;
    b++;
}
else
    f = false;
if (f)
    /* true body */
else
    /* false body */
```

Now the code is much cleaner and instrumentations are easier to made to it. Note that after this conversion, we get one more control flow path which is a infeasible path (the path passing through the false body of the first branch and the true body of the second branch). This is fine as the behavior of the program is not changed.

4.2 State savings on C++ variables

In the prior chapters, when we need a state saving, we normally call two functions: *store*() and *restore*(). But what is behind those two function calls? In this section, we will introduce how and where to store variables in Backstroke. Besides variables of basic types, we will also discuss how to correctly store a C++ object with class type, including the case where the object is referenced by a pointer or reference of its superclass type.

4.2.1 The storage stack

To perform state savings, we need a storage container that provides interfaces to perform the storing and restoring operations. Since the state saving made by Backstroke is incremental, the names and number of variables to be stored and the size of space needed are unknown at compile time. Therefore, in Backstroke we use a stack that stores all variables. As stack is a LIFO (last-in-first-out) data structure, a variable that is firstly saved and pushed will be popped lastly. And during the OPDES, reverse events are also implemented in a LIFO order (the first reverse event is used to remove the side effects produced by the last forward event in the sequence). Hence stack is the right container to store variables and we only need one stack for all event handlers.

To let our stack hold variables of arbitrary types, one way in C++ is that for each variable we store a pointer of `void *` type pointing to the copy of the variable, and during the restoration, the referenced variable should be casted to the proper type. However, when a great number of variables are stored, the memories possessing their copies may be scattered, and for each variable an additional size information is needed in the heap memory and is usually attached to the memory containing the variable, which can bring significant time and space overhead. To amortize this overhead, we could use a memory pool that allocates memories to store variables. In addition, we also observed that most stored variables are of basic types like integers and floating points. To decrease the memory used to store those variables, we create several special stacks which are used to store variables of basic types.

The storage stack should also provide an interface to clear the items inside. We can do this by popping all items and release the memory for each item. However, in some scenarios, instead of clear all items, we may only want to remove some old items but keep new ones, when it is more convenient if we can access the stack from both ends. In Backstroke, our storage stack has the type `std::deque<void*>`, which is a double-ended queue that satisfies our needs. The function *store*() and *restore*() are defined as below:

```
std::deque<void*> storageStack;

template<class T>
void store(const T& obj) {
    storageStack.push_back(new T(obj));
}

template<class T>
void restore(T& obj) {
    T* t = static_cast<T*>(storageStack.back());
    storageStack.pop_back();
    obj = *t;
    delete t;
}
```

4.2.2 Storing C++ objects

For a variable with basic types or POD (plain old data) types, there are no side effects when copying it during state savings. However, for a C++ object of class type, its copy constructor, assignment operator, and destructor (we will call them *big three functions* below) are triggered when storing and restoring it. Backstroke requires that those big three functions should be defined "correctly". Here we discuss the correctness of them.

Given an object *obj* of type *class S* in the original program, if it is stored and restored in the forward and reverse programs, we have to make sure the state is recovered after running the reverse program comparing to that before running the forward program. We store an object by calling its copy constructor to create a copy of it which is stored in our storage stack. We don't use assignment operator here because the class of the object may not have a default constructor, in which case creating an object is not that straight forward. When we restore this object, normally the object is already there, and we can just assign the copy to it by calling the assignment operator. After the restoration, we delete the stored object that calls its destructor. We should guarantee that after storing and restoring an object the simulation state is correctly rolled back. Below are the *store*() and *restore*()

implementations as shown before with comments showing when those three functions are called.

```
template<class T>
void store(const T& obj) {
    storageStack.push_back(new T(obj));  // The copy constructor is
        called.
}

template<class T>
void restore(T& obj) {
    T* t = static_cast<T*>(storageStack.back());
    storageStack.pop_back();
    obj = *t;  // The assignment operator is called.
    delete t;  // The destructor is called.
}
```

To guarantee the correctness, we require that there is no side effect in the big three functions. For instance, each of them should not modify a global or static shared variable that belongs to the simulation state and could affect the simulation result. In addition, the copy constructor and assignment operator usually should make a deep copy of the object. This means if this object has a pointer pointing to another chunk of memory that is managed by this object, this piece of memory should also be copied in those two functions, and needs to be released in the destructor. An example is `std::vector`, whose copy constructor and assignment operator both make a deep copy of the array inside. However, this requirement may be too strong as sometimes it could conflict with the behavior of the original program. An apparent situation is that the copy constructor does not alway make a deep copy of itself when several instances of this class have links to the same data and any one can update it. If the copy constructor makes a deep copy, the behavior of the program would be changed. In the next part we will propose a solution to handle this issue.

4.2.3 Storing an object referenced by a pointer/reference with its superclass type

In C++, an object with a pointer or reference type *Base* may have a concrete type *Sub* which is a subclass of *Base*. However, at compile time, it is impossible to identify the real type of such an object. As a result, calling the copy constructor of the *Base* class will not do the whole copy of that object, but only the part that belongs to the type *Base* only. This will lead to the error in state savings as shown below:

```
class Base {
    int baseVal;
};

class Sub : public Base {
    int subVal;
};

Base *p = new Sub;
store(*p);  // Error! Only p->baseVal is stored.
```

Polymorphism is the solution to this problem. What we need is the virtual versions of the big three functions. However, in C++ only the destructor can be declared as a virtual function.

To get rid of this restriction, we can define our own virtual versions of the three big functions. Specifically, we require that the class *Base* should provide two special interfaces as virtual functions: a *clone* function that returns a copy of the object; an *assign* function that assigns itself to another object.

```
class T {
public:
    virtual T* clone();
    virtual void assign(T* obj);
};
```

```

class S : public T {
public:
    virtual T* clone()
    { return new S(*this); }
    virtual void assign(T* obj)
    {  *this = *(dynamic_cast<S*>(obj)); }
};
```

There are two advantages of those two new interfaces: first, they are provided as the complementary interfaces to the classes in the original program and hence will not change its behavior, and when the original big three functions cannot fulfill our requirement, we can always defines those two virtual functions; second, in this way we could correctly store an object referenced by a pointer or reference of its superclass.

The code below shows the new *store* and *restore* functions using the new interfaces.

```
template<class T>
void store(const T& obj) {
    storageStack.push_back(obj.clone());  // The virtual copy
        constructor is called.
}

template<class T>
void restore(T& obj) {
    T* t = static_cast<T*>(storageStack.back());
    storageStack.pop_back();
    obj.assign(*t);  // The virtual assignment operator is called.
    delete t;  // The destructor is called.
}
```

Note that it is possible that the original destructor could not release all resources generated by the *clone*() function. In this case, the *assign*() function should release those resources so that after running *assign*() and destructor, all resources generated by *clone*() will be safely released.

4.3 Handling function calls

In this section, we describe various techniques for extending our program inversion framework to handle programs with function calls. In addition, we discuss the pros and cons of these techniques.

4.3.1 Inlining

A nave technique of handling multiple function programs is to inline all the functions in the program such that program is effectively converted to a single function. The existing technique can then be applied to such a program to generate the corresponding forward program P^+ and inverse program P^-. However, there are two major drawbacks of this approach: 1) In programs with recursive calls, inlining can lead to a infinitely large program and thus, recursion needs to be handled in a special manner, 2) Even without recursion, inlining causes a large increase in the program size. Consequently, the corresponding VSG generated for this program is a very large graph. Performing the lowest-cost search on this graph for generating the RG then becomes practically infeasible. However, inlining does ensure that the most optimal solution compared to the other techniques described later.

4.3.2 State Saving

Another naive but practical approach is to save the incoming state before any function call. As a result, when generating the program inverse, it is possible to ignore the function call completely and restore the state saved before the call to the function. Although this method ensures correct programs and allows the application of the existing techniques, it negates the very purpose of generating inverse programs. By forcing a state save before every function call, this approach incurs a large cost.

4.3.3 Unique VSG and unique RG for each method

In this approach, we generate a unique VSG for each non-virtual method. Further, considering the input parameters as the input state I and the return value of the method as the output state O, we generate a RG for this method. Consider a method $foo()$. In the VSG of the callee of $foo()$, there exists a node which represents the call to $foo()$. This node is connected to the return value via an incoming edge and has outgoing edges to the parameters of the called method. Using this approach, we add a new node in the VSG of the callee representing a call to $foo()^-$. The parameters of $foo()$ are connected to $foo()^-$ via edges incoming into $foo()^-$ while $foo()^-$ is connected via an outgoing edge to the return value of $foo()$. Further, the cost associated with these edges is as determined by the RG generated for $foo()$. This method allows a scalable approach to generating forward and reverse programs. Although the final program generated by inlining carries a lower cost than by this approach, it is possible to use this approach in a practical setting. However, this approach suffers from one major drawback. Suppose that a parameter of $foo()$ is overwritten just before the call to $foo()$. The value of the parameter needs to be stored in the forward program P^+ since it is can not be recovered by computational methods. However, if the reverse program use $foo()^-$, it redundantly recovers the parameter value at the call to $foo()$ only for the recovered parameter value to be replaced by the stored value. This lack of flexibility in choosing the recovery mechanism for each parameter individually and being forced to use the same mechanism for all parameters leads to a sub-optimal solution.

4.3.4 Unique VSG and multiple RG for each method

This approach resolves the drawback in the previous approach. We still generate a unique VSG for each method. However, a separate RG is generated for each parameter with the parameter as the input state I and the return value of the method as the output state O. The separate RGs are then used to generate multiple inverse versions of the original method, with each inverse version generating a single parameter as the return value. Consider a method $foo()$ with parameters a and b and return variable c. Using this approach, we generate the inverse versions $foo()A^-$ and $foo()B^-$. In the VSG of the callee to $foo()$, there are edges from c to $foo()$, from $foo()$ to a and from $foo()$ to b. We additionally add edges from a to $foo()A^-$, b to $foo()B^-$, $foo()A^-$ to c and $foo()B^-$ to c. This allows us to make independent decisions when generating the reverse code for each parameter of $foo()$.

4.4 Handling STL containers

C++ STL containers are commonly used in the real code, including the simulation programs. From high level, if we take an STL container instance as a state variable, and if it is modified in the event function, we have to use state saving technique that stores and restores this container. From the respect of safety, if the type of the container elements has a proper copy constructor and a copy assignment operator, the copy constructor and copy assignment operator of STL container will call it repeatedly on each element to store and restore the whole container; from the respect of performance, this method suffers from bad efficiency: even if only one element in the container is modified, we have to copy the whole container during state saving. Due to the restrictions of our method, we cannot handle the C++ STL container from low level (mass aliasing exists in STL container). In this section, we propose a method to "undo" some modifications on STL containers better than state saving, with the help of high level information.

As a basic example, let take a look at the most commonly used STL container: `std::vector`. A `std::vector` behaves like an array, except its size can by adjusted dynamically at runtime. It has an interface named `push_back()` which adds an element to the end of the vector, and hence the size of it is incremented by one. If a vector as a state variable is changed in this way, it is apparent that we can call its another interface `pop_back()` to undo this modification - a much more efficient way than state saving. Now let's consider how to undo a `pop_back()`. Note that since the last element of the vector is popped and destroyed, we cannot call `push_back()` to undo the pop unless we have a copy of the popped element. Here we come back to our forward-then-reverse function call approach. In the forward function of `pop_back()`, we make a state saving on the being popped element before popping it, and in the reverse function of `pop_back()`, we can just restore the saved element and then call `push_back()` to add it back to the vector. The forward and reverse functions of `push_back()` and `pop_back()` are generated by hand but can be recognized by Backstroke. They are shown below:

```
template <class T>
inline void bs_vector_push_back_forward(std::vector<T>& v, const T& t) {
    v.push_back(t);
}
template <class T>
inline void bs_vector_push_back_reverse(std::vector<T>& v) {
    v.pop_back();
}
template <class T>
inline void bs_vector_pop_back_forward(std::vector<T>& v) {
    store(v.back());
    v.pop_back();
}
template <class T>
inline void bs_vector_pop_back_reverse(std::vector<T>& v) {
    T t;
    restore(t);
    v.push_back(t);
}
```

Now let's consider `std::queue`, which is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure. It has two interfaces that modify the internal data: `push()` and `pop()`. `push()` pushes an object to the end of the queue and `pop()` pops the object at the front of the queue. However, there is neither an interface that could push an object to the front, nor an interface that could pop an object from the end. Hence it is impossible to write "cheap" reverse functions for `push()` and `pop()`. But an `std::queue` actually contains a real container inside, and the default one is a `std::deque`, which has more interfaces that we need. Therefore, we can simply change the `std::queue` into a `std::deque` then use the forward/reverse functions of it.

The Table 2 shows how we generate the forward and reverse code for common STL containers and their interfaces.

Table 2: The forward and reverse methods for some interfaces of STL containers

STL container & interface	**Forward Code**	**Reverse Code**
vector::push_back() deque::push_back() list::push_back().	Call push_back().	Call pop_back().
vector::pop_back() deque::pop_back() list::pop_back().	Store the popped element, and then call pop_back()	Restore the saved element and call push_back() to push it to the container.
vector::insert() deque::insert() list::insert()	Store the index/iterator of the element to be inserted and call insert().	Remove the element with the saved index/iterator by calling erase().
vector::erase() deque::erase() list::erase	Store the element to be erased and its position, then call erase().	Restored the erased element and insert it to the saved position.
set::insert() map::insert()	Call insert() and store the iterator of the inserted element.	Remove the stored element by calling erase().
set::erase() map::erase()	Store the element to be erased can call erase().	Insert the stored element by calling insert().
stack::push()	Call push().	Call pop().
stack::pop()	Store the popped element, can then call pop().	Restore the saved element and call push() to push it to the stack.
queue::push()	Use a deque instead of a queue, and call deque::push_front().	Call deque::pop_front() to pop the element at the end of the queue.
queue::pop()	Use a deque instead of a queue. Save the popped element, and then call deque::pop_back().	Call deque::push_back() to push the saved element to the front of the queue.

CHAPTER V

CONCLUSION AND FUTURE WORK

In this thesis we presented a compiler framework Backstroke that generates forward and reverse programs for a given program automatically. Two novel intermediate representations are critical in Backstroke: a VSG shows equalities between values in the program, and each equality is constrained by a set of control flow paths; a RG is the search result of the VSG which shows all data dependences in the reverse program and the forward and reverse programs are generated from the RG.

After the VSG is built and all target and available nodes are located, we start a search from target nodes until available nodes are reached. A special state saving node is added to the VSG such that any value could be retrieved at least using the state saving method, though normally it is more expensive than other methods. The search algorithm tries to get the search result with the least cost - that is, minimize the usage of state savings. It also guarantees each target value is retrieved on all control flow paths in the program. To deal with arrays, we introduce special array nodes into the VSG and show equalities between subregions of arrays. An array subregion is a subset of array elements, and during the search for a specific subregion, we perform set operations on subregions to make sure the target subregion of the array is retrieved. To retrieve an array subregion from a loop, we developed a demand-driven algorithm that retrieves one array element from each iteration of the loop.

To handle loops, we first transform the loop into a single-entry single-exit loop like a while loop or do-while loop. Given a while loop, the key idea behind retrieving an input value of the loop from its outputs is to retrieve the corresponding input value in each iteration. We need to augment the VSG with additional edges that connect the loop's input and output values to the input and output values of the body. We may then run the search procedure, which yields an RG with cycles. The presence of cycles indicates that we need to build loops in the reverse program. To construct such a loop in the reverse program, we also build a correct loop condition that guarantees the loop in the reverse program has the same number of iterations at runtime as the loop in the original program.

The Backstroke compiler is implemented based on the open-source ROSE compiler [21] for C++ programming language. Specifically, Backstroke is a subproject of ROSE and it can be downloaded from http://www.rosecompiler.org (Backstroke is located at ROSE/projects/backstroke).

Future work We think there are two directions to the future work: the first one includes eliminating restrictions in our method; the second one is building a framework for program inversion cooperating with our compiler.

We could expand the scope of the target language by eliminating restrictions in our method. One main limitation is that we cannot handle programs with aliasing. Since our method heavily depends on SSA, we may need an SSA representation to resolve aliases in the program [14]. In addition, we could also try to handle linked data structures. We have seen previously that the method to handle arrays can be applied to linked data structures, but more informations are required from more sophisticated analyses. For example, from shape analysis, we may detect a pattern of linked list traversal, and then we know each list node is traversed only once - an important information that is very useful. Another limitation of our method is that we lack of ability to solve equations to retrieve some values. For example, assume we have $a = x + y$ and $b = x - y$, then by solving equations we could get x and y from a and b through $x = (a + b)/2$ and $y = (a - b)/2$. How to combine this capability to our VSG is a potential problem to be solved.

In Chapter 4, we have seen that some high level informations are helpful for Backstroke to generate better result. Also, we have also seen that to properly perform state savings on C++ objects, we need three new interfaces for us to do the clone, copy and destroy. This suggests a framework for program inversion, in which some constructs are provided to implement some necessary functions, or give the compiler some hints to generate better result (think about how we handle C++ STL containers). From another aspect, this new framework provides new language features that could be recognized by Backstroke. We believe that this new framework (or new language) together with Backstroke can produce much better results for program inversion.

REFERENCES

[1] Abramov, S. and Glück, R., "The Universal Resolving Algorithm: Inverse Computation in a Functional Language," *Science of Computer Programming*, vol. 43, no. 2-3, pp. 193–229, 2002.

[2] Akgul, T. and Mooney III, V. J., "Assembly instruction level reverse execution for debugging," *Thesis*, vol. 13, pp. 149–198, Apr. 2004.

[3] Alpern, B., Wegman, M. N., and Kenneth, F., "Detecting Equality of Variables in Programs," *PPL*, no. January, 1988.

[4] Ball, T. and Larus, J. R., "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1319–1360, July 1994.

[5] Ball, T. and Larus, J. R., "Efficient Path Profiling," in *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'96)*, pp. 46–57, 1996.

[6] Biswas, B. and Mall, R., "Reverse execution of programs," *ACM SIGPLAN Notices*, vol. 34, pp. 61–69, Apr. 1999.

[7] Bodik, R., Gupta, R., and Sarkar, V., "Abcd: Eliminating array bounds checks on demand," *PLDI*, 2000.

[8] Briggs, J. S., "Generating reversible programs," *Software: Practice and Experience*, vol. 17, pp. 439–453, July 1987.

[9] Carothers, C. D., Bauer, D., and Pearce, S., "Ross: A high-performance, low-memory, modular time warp system," *Journal of Parallel and Distributed Computing*, vol. 62, no. 11, pp. 1648–1669, 2002.

[10] Carothers, C. D., Perumalla, K. S., and Fujimoto, R. M., "Efficient Optimistic Parallel Simulations Using Reverse Computation," *PADS*, 1999.

[11] Cooper, K. D., Simpson, L. T., and Vick, C. a., "Operator strength reduction," *ACM Transactions on Programming Languages and Systems*, vol. 23, pp. 603–625, Sept. 2001.

[12] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K., "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[13] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K., "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[14] Cytron, R. and Gershbein, R., "Efficient accommodation of may-alias information in ssa form," in *ACM SIGPLAN Notices*, vol. 28, pp. 36–45, ACM, 1993.

[15] Glück, R. and Kawabe, M., "Revisiting an automatic program inverter for Lisp," *ACM SIGPLAN Notices*, vol. 40, pp. 8–17, May 2005.

[16] Hou, C., Quinlan, D., Jefferson, D., Fujimoto, R., and Vuduc, R., "Synthesizing loops for program inversion," *4th Workshop on Reversible Computation*, 2012.

[17] Hou, C., Vulov, G., Quinlan, D., Jefferson, D., Fujimoto, R., and Vuduc, R., “A new method for program inversion,” *Compiler Construction*, 2012.

[18] Jefferson, D. R., “Virtual time,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[19] Kawabe, M. and Gl, R., “The Program Inverter LRinv and Its Structure,” pp. 219–234, 2005.

[20] Muchnick, S. S., *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[21] Quinlan, D., “Rose: Compiler support for object-oriented frameworks,” *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[22] Riley, G. F., “Simulation of large scale networks ii: large-scale network simulations with gtnets,” in *Proceedings of the 35th conference on Winter simulation: driving innovation*, pp. 676–684, Winter Simulation Conference, 2003.

[23] 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.

[24] Ross, B. J., “Running programs backwards: The logical inversion of imperative computation,” *Formal Aspects of Computing*, vol. 9, pp. 331–348, May 1997.

[25] Rus, S., He, G., and Rauchwerger, L., “Scalable array ssa and array data flow analysis,” *Languages and Compilers for Parallel Computing*, pp. 397–412, 2006.

[26] Sebastian Pop, Pierre Jouvelot, and Georges-Andre Silber, “In and Out of SSA : A Denotational Specification,” *Static Single-Assignment Form Seminar*, 2009.

[27] Srivastava, S., Gulwani, S., Chaudhuri, S., and Foster, J. S., “Path-based inductive synthesis for program inversion,” in *PLDI ’11*, ACM Press, 2011.

[28] Vulov, G., Hou, C., Vuduc, R., Fujimoto, R., Quinlan, D., and Jefferson, D., “The backstroke framework for source level reverse computation applied to parallel discrete event simulation,” in *Proceedings of the Winter Simulation Conference*, pp. 2965–2979, Winter Simulation Conference, 2011.

[29] Wolfe, M., “Beyond Induction Variables,” in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation (PLDI)*, 1992.