Backstroke Project Report 2010/2011

George Vulov and Richard Fujimoto
School of Computational Science & Engineering
Georgia Institute of Technology

BIBTEX:

document created on: September 14, 2011
created from file: "Backstroke Report".tex
cover page automatically created with CoverPage.sty
(available at your favourite CTAN mirror)

Backstroke Project Report 2010/2011

George Vulov* Richard Fujimoto*
* School of Computational Science & Engineering, Georgia Institute of Technology

I. Overview of Backstroke work for the 2010/2011 contract year

During the 2010/2011 school year, we explored a number of directions for the Backstroke project, including both improving the power of its inversion methods and experimenting with applying Backstroke to real simulation systems. The different areas of work a briefly tabulated below, with more detail in subsequent sections.

1) Implementation of regenerative inversion (section III): Regenerative inversion algorithms (described in section II-B7) are powerful methods for reducing state saving requirements by dynamically recomputing lost values rather than saving them. We implemented the state-of-the-art inversion algorithm described by Akgul & Mooney [1], [2], [3] as part of the Backstroke framework.
2) Extending the ROSE compiler framework with new analysis (section IV): During Backstroke development, it became apparent that the analysis tools present in the ROSE compiler framework were not sufficient for Backstroke's needs. For this reason, we implemented a novel static single assignment analysis within the ROSE compiler framework; this analysis has since been adopted for use by other ROSE-based projects.
3) Implementation of automatic checkpointing (section V): Checkpointing is an important tool for reverse computation. When the state of a function is small but its computation time is long, checkpointing is always the most efficient inversion method. Furthermore, checkpointing can be used as a comparison baseline for comparing with other inversion methods and can serve as a debugging tool that checks the accuracy of other inversion methods. We implemented automatic checkpointing for functions based on interprocedural analysis.
4) Experimental study in automatic parallelization of self-federated simulations (section VI): One approach to building parallel models involves instantiating multiple instances of a sequential model and integrating (federating) them together. For example, a large computer network can be simulated by partitioning the network into subnetworks, instantiating a sequential simulator for each subnetwork, and federating the resulting sequential simulation models. This parallelization methodology that involves self-federating an existing sequential simulation code has been successfully applied to create large-scale telecommunication network simulators [4], among other applications. We proposed and tested a methodology for making existing sequential models suitable for optimistic execution by employing a reverse compiler (namely Backstroke).
5) Experimental study in applying Backstroke to the GTNetS (section VII): The Georgia Tech Network Simulator (GTNetS) [5] is a large full-featured discrete event simulator of computer networks. The simulator consists of over 200,000 lines of real-world C++ code and simulates all layers of the network stack. We chose to apply Backstroke to GTNetS as a real-world stress test that would reveal shortcomings in the inversion methods currently implemented and motivate further Backstroke development.

The work areas outlined above are discussed in more detail in the following sections; first we begin with an overview of the literature related to reverse computation.

II. Literature about Reversible Computing

A. Applications of reverse execution and program inversion

1) Optimistic Discrete Event Simulation: Distributed discrete event simulations consist of individual discrete event simulators sending timestamped messages to each other over a network interface. Optimistic distributed simulation algorithms, such as Time Warp [6] allow nodes (logical processes) to simulate forward without guarantees that they will not receive any events in their past. If a logical process receives a message timestamped prior to its current simulation time, it must restore its state to the previous time (roll back), before processing the message in question. Traditionally, rollbacks have been implemented using variations of state saving; however, reverse execution techniques have been shown to offer significant speed and memory advantages for large distributed simulations [7], [8], [9]. One approach is generating rollback code automatically from the simulation model source code [7]. It is also possible to use domain knowledge about the simulated model to construct reverse event codes [8], [9]. Furthermore, if the underlying model is perfectly reversible, Time Warp's storage requirements for rollback can be relaxed [10], [11]; Baker was the first to note the connection between Time Warp and reversible processes [12].

2) Debugging: Debuggers are very good at pausing a program, examining its state, and stepping forward; however, a flaw is always executed before it is manifested. Developers have to resort to the time-consuming and potentially difficult process restarting the program to reproduce a flaw. If debuggers offered the ability to easily run programs backwards, the debugging process would be greatly enhanced. Work on reversible debuggers started in the late 1960s with [13], [14],

and new methods to reverse programs efficiently have been developed continuously since then [15], [16], [17], [18], [19], [20], [3], [21], [22]. The popular open source debugger GDB has also has some support for reverse debugging.

Debugging applications have different granularity requirements for reverse execution than optimistic discrete event simulation. In optimistic discrete event simulation, the unit of inversion is the execution of a single event (which may involve a lot of computation). On the other hand, debuggers generally must be able to step backwards in small steps, often a single instruction.

3) Undo in applications: Undo functionality in interactive applications [23] is an indispensable tool for usability and is now present in almost all interactive environments. Undo functionality is naturally related to reverse computation and can be implemented with state saving or using more advanced reverse execution techniques [23]. Briggs [24] implemented the undo functionality of a commercial cricket scoring system by automated inversion of each event.

4) Fault Tolerance: The detection of data corruption is an important problem in the study of fault-tolerant systems. Bishop [25] proposes using reverse execution to check for corruption errors — executing a function in reverse must produce the original inputs of the function and match the intermediate values of the forward execution.

5) Automatic parallelization of list operations: List homomorphisms [26] are a generalization of the list functions that can be efficiently implemented with the functional operators *map* and *reduce*. List homomorphisms are naturally parallelizable and can implement many common list operations, such as finding the sum, length, or maximum of a list. Morita et al. [27] describe a method for deriving a list homomorphism given two sequential codes implementing the operation: by accumulating left-to-right and by accumulating right-to-left. It is a theoretical result that a list homomorphism exists if it can be implemented by accumulating left-to-right and vice-versa; however, for many problems, writing the sequential code is much easier than deriving the list homomorphism. Morita et al. reduce the problem of finding the list homomorphism to that of finding a type of inverse to the input sequential programs; if the inversion succeeds, their approach offers automatic parallelization.

6) Education and Visualization: When visualizing the execution of a computer program, the ability to run it in either the forward or reverse directions is highly desirable. In student testing of the DYNAMOD program animator [28], the ability to animate the program in reverse was a highly requested feature. Subsequently, reverse program execution was included in the DYNALAB program visualization environment [29]. Another software visualization project to take advantage of reverse execution is the LEONARDO software visualization environment [30]. Whereas DYNALAB was focused on helping computer science students, LEONARDO allowed visualizing arbitrarily complex C programs. When visualizing larger programs, the efficiency of the reverse execution becomes very important.

7) Synchronizing structured data: An interesting application of program inversion is in the synchronization between two different representations of the same information [31], [32], [33]. A common example is a document-view implementation in a text editor — changes to the document should be reflected in the view and vice-versa. The approach taken by Mu, Hu, and Takechi [31] is to define the transform from the document to the view in a special language that can only express injective functions. Any program written in such language is invertible, and the inverse can be used to transfer changes from the view back to the original document.

8) Backtracking: Backtracking [34] is an approach for finding solutions to combinatorial problems. The solution to the problem is built incrementally by finding ways to extend a partial candidate solution. If it is discovered that the candidate solution cannot lead to a lead to a correct solution, the algorithm "backtracks", and continues the search from a previous partial candidate. Floyd [35] demonstrated how a backtracking algorithm can be written without any reference to the search strategy, only specifying the choice points. The actual search is performed by generating dual forward and reverse codes from the generic source. The forward code performs search and stores necessary information, while the reverse code implements backtracking by reversing the effects of the forward code. Floyd recognized that certain operations, such as incrementing an integer, are reversible and hence do not require state saving.

9) Automatic Differentiation: Reverse computation has applications to the reverse accumulation algorithm for automatic differentiation. Automatic differentiation is an approach of computing derivatives of a function directly from the source code that implements that function [36]. The building blocks of any function implemented in code are arithmetic operations and common math functions such as sine and cosine; these primitives have well-known derivatives. The derivative for the overall function can be calculated by tracing the calculation of the function from these known primitives and applying the chain rule at each step. Reverse accumulation starts with the final function value and applies the chain rule backwards through the code; it requires accessing all intermediate values of the computation in reverse order. The simplest implementation of reverse accumulations is recording a full execution trace of the function evaluation [37]. A strategy for reducing the memory requirements of this approach is checkpointing and processing a single section of the computation at a time [38], [39], [40].

B. Program inversion

A program P can be viewed as a function $P(x) = y$ where x is the input to the program and y is its output. If P is injective, there is exactly one input value for every output of P. For injective programs, program inversion is the problem of finding a program P^{-1} that computes P's input given P's output; namely $P^{-1}(P(x)) = x$ for all x. If P is not injective, there are two flavors of program inversion: one approach is to find *all* inputs to P that produce a given output value y;

another approach is to augment P's output so as to make it injective. Note that for any program P, the program $P'(x) = (P(x), x)$ is injective and has a trivial inverse; storing the input to the program is always enough to guarantee invertibility.

McCarthy showed that the inverse of a Turing machine is computable when it exists, but noted that the existence of an inverse itself is undecidable [41]. Consequently inversion algorithms that do not augment the output of the original program either only work for a subset of all possible programs or do not always terminate.

1) Manual Inversion: Dijkstra was the first to demonstrate a program inversion of injective programs by hand, using postconditions to guide the program transformation [42]. Gries described rules for inverting a imperative programs that included branching and looping [43]. Gries supplied a postcondition for each branching statement, which could be used as the branching condition in the reverse direction. The "do" loop is inverted by deriving two conditions — a precondition which is false in the first iteration of the loop, and an exit condition which is false only for the last iteration of the loop. These constructs are identical to the ones later implemented in the reversible programming language Janus [44], [45], although Gries's work is not mentioned by the Janus authors. Chen and Udding formalized Gries's inversion rules with the intent of formally proving the correctness of inverse programs [46].

2) Korf-Eppstein Program Inverter: The Korf-Eppstein method for inverting functional programs uses a state-space search approach to program inversion. It starts with the forward program and transforms it until the inverse is reached. The working algorithm was first presented by Eppstein [47] and Korf [48]; it was later revisited (an named) by Glück and Kawabe [49]. The approach only works for programs that are injective. The inverter always terminates, hence it is not guaranteed to find a program inverse even if one exists. The actual class of invertible programs depends on the state space search strategy implemented and on heuristic portions of the algorithm.

The inversion process proceeds by operating on a set of facts. Each fact contains preconditions and an equation that is true whenever the preconditions are true. The initial state asserts that the function to be inverted is equal to its body, with no precondition. A terminal state also has no preconditions has all the input variables equal to functions of the output variables. Figure 1 demonstrates the inversion process for a function `f`, which adds 1 to its input if its input is odd, and subtracts 1 from its input if its input is even.

There are four operators that modify the set of facts: conditional expansion, precondition replacement, expression inversion, and conditional contraction. The example in Figure 1 demonstrates each one, albeit in a very simple case. Conditional expansion takes a fact containing a conditional statement, such as `if`, and breaks it into multiple facts, one for each branch that could be taken. Each of the new facts has an extra precondition that is taken from the conditional statement. The next operator, precondition replacement, replaces the preconditions of a fact with equivalent ones that

Initial fact:
`{}` $\implies$ `f = (if (= 0 (mod n 2)) (- n 1) (+ n 1))`

Conditional expansion:
`{(= 0 (mod n 2))}` $\implies$ `f = (- n 1)`
`{not (= 0 (mod n 2))}` $\implies$ `f = (+ n 1)`

Precondition replacement:
`{not (= 0 (mod f 2))}` $\implies$ `f = (- n 1)`
`{(= 0 (mod f 2))}` $\implies$ `f = (+ n 1)`

Expression inversion:
`{not (= 0 (mod f 2))}` $\implies$ `(+ f 1) = n`
`{(= 0 (mod f 2))}` $\implies$ `(- f 1) = n`

Conditional Contraction:
`{}` $\implies$ `(if (= 0 (mod f 2)) (+ f 1) (- f 1)) = n`

Fig. 1: Example of Korf-Eppstein Program inversion. The initial state is an expression for the function `f` while the final state is an expression for the input value `n` given the output of `f`.

are functions of only the output variables. In effect, the precondition replacement operator finds postconditions of an `if` statement that can distinguish which branch was taken based on the output of the statement. The dual operator of conditional expansion is conditional contraction, which takes two facts with opposite preconditions and combines them into an `if` statement. Conditional contraction only accepts preconditions that involve only the output variables. The combination of conditional expansion, precondition replacement, and conditional contraction converts an `if` statement in the forward direction to one that executes in reverse. Finally, the expression inversion operator is the one that inverts individual statements; it has rules for handling each language construct.

The most challenging aspect of program inversion is inverting the conditional statements. For each branch of a conditional statement, a postcondition must be found that uniquely determines wether that branch was taken or not. The Korf-Eppstein method derives these postconditions by organizing all values in a class hierarchy and then deducing the class type of each branch of a conditional statement. If the two branches of a conditional statements produce values in different classes, then testing the output value's class membership is the desired postcondition. The system functions much like a normal data typing system, but with much more specific data types. For example, the inversion in Figure 1 would not succeed unless odd integers and even integers were in different classes. The postcondition inference is a heuristic and is not guaranteed to find a suitable postcondition even if one exists. The power of the postcondition inference, along with the state-space search

strategy, define the class of programs invertible by a particular implementation of Korf-Eppstein.

3) LRinv: program inversion based on parsing theory: LRinv is another program inverter for injective functional programs [50], [51]. LRinv converts the original input program to an intermediate language that has a grammar-like structure. The corresponding grammar is easily inverted to produce a grammar that matches the output of the program given its input; unfortunately this grammar is usually nondeterministic. Eliminating nondeterminism from the inverse program is analogous to finding suitable disjoint postconditions for branching statements, so that during reverse execution the correct reverse branch can be taken. Kawabe and Glück eliminate nondeterminism from the inverse grammar by generating a deterministic LR parser for the grammar; if a deterministic version of the inverse grammar is found, the deterministic grammar can be translated back to functional form to produce the inverse of the original program.

4) The Universal Resolving Algorithm: The Universal Resolving Algorithm (URA) performs inverse computation on arbitrary (non-injective) programs written a the first-order functional language [52], [53]. Given a program P and its output y, URA returns all values x such that $P(x) = y$. Technically, URA is an inverse interpreter, rather than a program inverter, since no program P^{-1} is generated. Nevertheless, given an inverse interpreter and a forward program, partial evaluation of the interpreter with respect to the program source yields the desired inverse program; this is an application of the first Futamura projection [54]. Since URA works for arbitrary programs and inversion in general is undecidable, URA is not guaranteed to terminate - even when the set $P^{-1}(y)$ is finite and all its entries have been calculated by URA.

The applicability of the Universal Resolving Algorithm is not limited to functional languages. Let *Int* be an interpreter for an arbitrary programming languages, written in a first-order functional language. Then, URA can be applied to *Int* to perform inverse computation in the new language. Partial evaluation of URA with respect to *Int* yields an inverse interpreter in the language. approach was explored in [55] to create and test inverse interpreters for several languages, including a subset of the Java bytecode.

The inversion process for a specific TSG program P begins with the specification of the output of the program, Y_P, and the search space of possible input values, X_P. All values in TSG are S-expressions and atoms (akin to Lisp). Hence, URA represents a general set of values as an S-expression with variables coupled with inequalities on the variables. For example, $\langle [a_1, a_2], \{a_1 \neq a_2, a_2 \neq \text{`}A\}\rangle$ represents all pairs of atoms in which the first atom is not equal to the second and the second atom is not equal to A. The input to URA is the program code P, its output class Y_P, and the input class X_P. Classic function inversion can be achieved by making the output class Y_P a single concrete value, while letting the input class consist of general variables with no restrictions.

Inversion proceeds in three stages: generation of a process tree for the input program, tabulation of input classes and the corresponding output classes, and traversal of the table to extract valid input classes for the given output class. The process tree is generated by executing the program with its partially specified input. When a conditional statement is encountered, two branches are added to the process tree, signifying the two possible paths of the computation. The input class is also partitioned into two disjoint sets based on the conditional test, and each input class is associated with the appropriate branch of the conditional. Continuing this process until a terminal branch yields an output expression y_i along with a corresponding input class x_i that would produce the output. Tabulation is the process of traversing the process tree and producing a table of the (x_i, y_i) input-output pairs. Since the the process tree can be infinite, tabulation must traverse the tree in breath-first order. Finally, each input-output pair (x_i, y_i) is unified with the (X_P, Y_P), the input search space and the output class; if the unification succeeds the result is printed. Unless the inverse is a single value, the results of the inversion will generally contain free variables restricted by inequalities.

5) Inverting Imperative Programs Through Logic Programming: Inversion can be naturally formulated in a logic programming language, such as Prolog, since the semantics of a program are declarative. Ross demonstrates a method of converting an imperative program into a logic program from which the inverse of the original can be inferred [56]. The ability of logic programming languages to return multiple results from a single query allows this approach to invert non-injective programs. The power of the approach depends on the inference strategy used by the underlying logic programming language. Since complete inference is undecidable, logic programming languages implement limited inference strategies.

A logic program consists of a set of a set of implications (Horn clauses) of the form $H \Leftarrow B_1 \wedge \cdots \wedge B_n$. The head H can be empty, in which case the clause is known as a fact. A query has the form $B_1 \wedge \cdots \wedge B_n$; the inference engine searches for values of the variables that make the query true and returns them. An imperative program P can be converted into a predicate $P(X, Y)$ that specifies the relation between the input and output values of the program. Figure 2 has an example of an imperative `while` loop and its equivalent logic representation. A query that gives the initial state and asks for the final state results in an execution very similar to that of the imperative program, with iteration being replaced by tail recursion.

Unfortunately, the straightforward conversion from an imperative program to a logic program illustrated in figure 2 cannot be used for inverse queries. Common inference strategies, such as Prolog's left-to-right goal selection, are tuned for inference in one direction. Ross's contribution in [56] is introducing inverted logic semantics for imperative programs. Once an imperative program is transformed into a logic program according to these inverted semantics, commonly used inference strategies can be used for inverse computation.

6) Restorative incremental inversion of imperative programs: This class of approaches apply to inversion of im-

Imperative while loop:

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

Logic Program (forward semantics):

```
while([Y, M, N], [Y, M, N]) :-
    not N > 0.

while([Y, M, N], Result) :-
    N > 0,
    N2 = N - 1,
    Y2 = Y + M,
    while([Y2, M, N2], Result).
```

Fig. 2: An imperative `while` loop and the equivalent Prolog code that executes it in the forward direction. Querying `while([0, 2, 2], Result).` yields `Result = [4, 2, 0]`.

perative programs that are not necessarily injective. The imperative program is essentially executed in reverse, with each modifying operation in the original execution being undone individually; the overall side-effects of the program are thus reversed *incrementally*. If an operation is not locally reversible, the modified state is explicitly saved during the forward execution and restored during reverse execution; hence the name *restorative* inversion. Floyd [35] describes taking a forward program and creating a pair of programs from it - an instrumented forward program that saves lost state and the corresponding reverse program to undo the actions of the forward program. Floyd recognized that some statements, such as variable incrementation, do not require additional storage to be reversed. Briggs [24] applied a similar technique to implement undo functionality in a commercial cricket scoring system. While Floyd performed the program conversions manually, Briggs used a generator to create the forward and reverse programs from the same uninstrumented forward code. Biswas and Mall [19] applied the same approach to C to create a debugger that could step in both the forward and reverse directions. Carothers, Perumalla, and Fujimoto [7] also generated inverses for C programs, but their approach, unlike that of Biswas and Mall, used compilation of the reverse code to achieve better performance.

The basic approach to incremental inversion is to invert each statement or machine instruction individually, and then put the inverted statements together with inverted code flow. A lossless (logically reversible) assignment statement, such as incrementing a variable, can be reversed simply by performing the opposite operation, decrementing the variable. In the C programming language, lossless assignment statements include +=, -=, and ^= (exclusive-or), given that the left-hand variable does not appear on the right-hand side. Figure 3 lists the rules for inverting a sequence of statements that do not include control flow statements.

Inverting control flow requires finding inversions for conditional statements, such as `if` and `switch`, and loop statements, such as `for` and `while`. If the value of the predicate of an `if` statement is unaffected by the code inside the statement, the same predicate can be used during the reversal to test which branch should be reversed. On the other hand, if an `if` statement potentially modifies the value of its predicate or the predicate has side effects, the branch taken must be stored explicitly. Storing the branch taken by an n-way `if/else` statement requires $\lceil \log n \rceil$ bits. Similarly, loops can be inverted by storing the number of times the loop body was executed. To undo a loop computation, the reverse loop body is executed as many times as the forward loop. If the number of iterations of the loop can be statically determined, no extra storage is needed to achieve the reverse control flow.

7) Regenerative incremental inversion of imperative programs: Regenerative incremental inversion is similar to restorative incremental inversion; it also applies to non-injective imperative programs and proceeds by reversing one operation at a time. However, reversing the side effects of a lossy operation does not necessarily require saving state during the forward execution; regenerative inversion algorithms attempt to recompute the missing values from other program values that are available. Such a method was first described by Akgul and Mooney [1], [2], [3]. Their work is specifically tailored towards assembly-level reverse code generation, but it extends naturally to imperative programs.

The heart of the Akgul and Mooney algorithm are the techniques used to reverse a single instruction within a basic block. A variable v is said to be defined by an instruction if that instruction overwrites v. If variable v is defined by the current instruction, the reverse code for the instruction should restore the previous value of v. The algorithm begins by finding all previous definitions of v within the program partition and enumerating all possible control flow paths from a previous definition of v to the current instruction. Reverse code is generated for each control path individually, and the reversals for the different paths are combined together with branching instructions that invoke the correct reversal for the actual path taken at runtime. There are two techniques for generating reverse code to recover v given its control path to a previous definition — the redefine technique and the extract-from-use technique. If the two techniques fail, the algorithm falls back to state saving.

a) The redefine technique: The idea behind the redefine technique is to execute the previous definition of v, thus restoring the value v had before it was overwritten. For example, if the previous modification of v was the statement `{v = a mod 15}`, and `a` has not been modified since then, executing that statement again restores the value of v. On the other hand, if a has been modified, the algorithm can first generate reverse code that restores the value of a, and then

Statement type	Original code	Instrumented code	Reverse code
Constructive assignment	x = x *op* a	x = x *op* a	x = x *op_inv*
Destructive assignment	x = a	t = a; x = a	x = t
Function call	foo(x,y)	foo(x,y)	foo_inv(x,y)
Statement sequence	s_1; s_2;	s_1; s_2;	inv(s_2); inv(s_1)

Fig. 3: Inversion rules of imperative program statements that do not include control flow constructs (from [7]).

apply the redefine technique. This process can be repeated recursively, restoring multiple variables in order to apply the redefine technique.

b) The extract-from-use technique: The extract-from-use technique takes advantage of the fact that the value of v is guaranteed to be unchanged for all instructions on the path between the current instruction and the previous definition of v. If some of those instructions have performed calculations using the value of v, the v might be recoverable from the values of other variables around it. For example if the statement `{a = b - v}` was executed before the previous definition of v, executing {v = b - a} would recover the value of v. If `a` or `b` have been modified, the algorithm could be recursively applied to recover their values before recovering the value of v.

The Akgul & Mooney inversion algorithm fails when multiple threads are present due to the statically unknown thread inter leavings. Lee proposes a regenerative incremental algorithm that works even in the presence of threading [57], [58]. Lee's algorithm records the thread interleaving points during the forward execution, and applies the extract-from-use and redefine techniques based on the actual interleaving taken during the execution. Because the reverse code depends on thread interleavings that cannot be determined statically, Lee's approach generates the reverse code dynamically during program execution.

8) Path-based inductive synthesis: Srivistava, Gulwani, Chaundhuri, and Foster have recently proposed a semi-automated approach for inverting injective imperative programs, named path-based inductive synthesis (PINS) [59]. PINS is considered a semi-automated approach for two reasons: first, it must be primed with a template for the reverse code; second, the inverse programs generated by PINS are not guaranteed to be correct — their correctness must manually be verified.

A PINS template consists of a program in which some of the expressions and predicates are unknown. The template also contains sets of concrete expressions and concrete predicates from which the values of the unknown ones may be filled. Inversion begins by appending the template inverse program to the original program; the combined program should produce the identity if the correct values for the missing template elements have been chosen. PINS maintains a constraint predicate that valid templates must satisfy; this constraint initially only restricts choices to programs that terminate. PINS applies an SMT solver to obtain template instantiations that satisfy the constraint; these template instantiations are symbolically executed. If the chosen template is correct, the output of the symbolic execution should equal the symbolic inputs; this assertion is added to the overall constraint and the refined constraint is used in the next iteration. Full algorithm details are presented in [59]. When PINS terminates, it produces a (small) number of template instantions that satisfy the constraint and thus are consistent as inverses with all the symbolic execution paths taken. The user of PINS must further verify that the template instantiations presented are indeed correct inverse programs. PINS fails to generate an inverse program if the template provided to it is not sufficiently general to describe the inverse program; in such cases the developer must refine the template and run PINS again.

9) Reverse execution with constraint solving: Recently Sauciuc and Necula have proposed a reverse execution approach for imperative languages based on constraint solving [22]. Sauciuc and Necula make the observation that assignment statements such as `x = x % 2` or `x = a + b` can be viewed as constraints on the variables in question. The dynamic execution path through the program induces a set of constraints relating initial values, intermediate values, and final values. Sauciuc and Necula then apply a constraint solver (also known as an SMT solver) to solve for the initial inputs of a program given its outputs.

The constraint solving approach does not guarantee that the computed intermediate and input values will be the same as the original input values; however the values computed are guaranteed to produce exactly the same output and follow the same execution path through the program. The intermediate values computed by the constraint solver can be brought closer to the actual values observed during the forward execution by recording extra intermediate values during the forward execution. Any value that is logged during the forward execution induces a new constraint that must be satisfied. Sauciuc and Necula propose using the constraint solver's internal logic to decide which bits to choose for state saving. If the constraint solver internally backtracks based on a bit of input, then saving that bit may reduce the search space of the constraint solver and produce a solution closer to the original input.

C. Reversible programming panguages

1) Janus: The programming language Janus was the first documented reversible programming language, created for a class project in Caltech [44], [45]. Janus is an imperative programming language that supports only integer arithmetic and allows no irreversible operations. The only assignment operators in the language are +=, -=, and ^= (xor assign), none of which discard any information. The Janus `if` statement

requires the programmer to supply an exiting condition whose truth value is equal to the branching condition of the `if` statement; in the reverse direction, the exit condition is used as the branching condition. The loop structure in Janus similarly requires two conditions: an entry condition that must be true only on the first iteration of the loop, and an exit condition that is true only for the last iteration of the loop. Janus has been formalized and proven reversible by Yokoyama and Glük ; they also developed a self-interpreter for Janus and made some observations about the practice of reversible programming [45]. Yokohama, Axelsen, and Glük also extended Janus with local variables, dynamic data structures, and parametrized procedure calls [60]. Mogensen has developed a partial evaluator for Janus that preserves Janus's reversibility properties [61].

2) The R language: R is another imperative reversible programming language, designed for use with the Pendulum reversible architecture [62]. It is very similar to Janus, although it is more restrictive in its controls flow primitives. All the updates are reversible, such as +=, -=, and ^=. Branching statements only have only one conditional expression, whose value must be the same before and after the branch is executed; in the reverse direction the same conditional expression is used for the branch. The only loop construct in the language is a for-loop; the expressions determining the bounds of the for-loop much evaluate to the same values before and after the loop.

3) Inv: Inv is a functional programming language that can only be used to define injective functions, i.e. functions that map distinct inputs onto distinct outputs [63]. Inv is a point-free language (arguments to functions do not appear explicitly) with relational semantics. Each Inv construct is invertible; notably the inverse of duplicating a value is an equality check. While only injective functions may be defined in Inv, it is not always clear how to derive an inverse given the forward function. In fact, the authors of Inv rely on backtracking in their implementation of the Inv interpreter, even though in theory the language is deterministic in both directions.

D. Theoretical results about reversible computing

1) Thermodynamic properties of reversible computation: Computation is ultimately performed on physical systems, which consume energy and dissipate heat during the computation. A computation can be viewed as a series of discrete steps, each of which modifies the existing state. Whenever the previous state of cannot be reconstructed from the current state, some information is lost (usually it is intentionally discarded). Landauer discovered that irreversible computation has to dissipate some heat — at least $kT \ln 2$ for each bit of information lost, where T is the ambient temperature and k is the Botlzmann constant [64]. Heat dissipation is often the limiting factor in processor clock speeds, so reversible computing holds the promise of creating hardware faster than currently possible [65]. Furthermore, due to conservation of energy, the heat dissipated by a computing apparatus is equal to the energy consumed by it; hence reversible hardware could potentially reduce power requirements for many battery-operated devices.

2) Reversible Turing machines: When Landauer discovered that irreversible computation causes heat dissipation, he postulated that irreversibility is fundamental to useful computation [64]. Later, Bennett showed this conjecture to be false by proposing a reversible Turing machines, a model for reverse computation that could reversibly simulate the computation of any standard Turing machine without producing waste output [66], [67]. A Turing machine performs computation by reading a tape symbol, writing a new value to the tape, and shifting the tape. The Turing machine is reversible if each read-write-shift tape operation has a shift-read-write which restores the previous state of the machine. A simple example of a non-reversible Turing machine is one that goes through and erases its input. On the other hand, a two-tape Turing machine that erases the input from its first tape while copying it to the second tape is reversible since every transition has an inverse. Any Turing machine can trivially be made reversible by adding a second tape that records a history of its computation. If were are only interested in the final result of the computation, however, such a machine would produce a tape full of unwanted (garbage) data and erasing a tape is a non-reversible operation.

Bennett showed that a reversible Turing machine can perform the computation of an arbitrary Turing machine while producing no garbage. First, an arbitrary Turing machine T can be made reversible by adding a second tape that records the history of the computation, as discussed previously. After the computation has halted the second tape can be erased simply by running T's reverse, T^R. Unfortunately the computation $T\,T^R$ leaves only the input and a blank tape - the output of T is itself lost. We can remedy this by adding a third tape and introducing a stage C to copy T's output to the third tape before running T^R. Note that C is a reversible operation since it only copies information to a blank tape, without destroying information at any step. Thus, the overall computation $T\,C\,T^R$ is both reversible and produces exactly the original input and T's output on its tapes.

The construction of reversible Turing machines can be analyzed in terms of the reversibility of the embedded computation. An arbitrary function $f : X \to Y$ can be embedded in an invertible function $f_R : X \to Y \times X$ by *state saving*, i.e. $f_R(x) = (f(x), x)$. Since Turing machines can compute non-invertible functions, any Turing-equivalent reversible model of computation must include some type of state saving. Bennett's contribution is showing that the saved bits required to make a computation reversible on the macro level are enough to make it reversible on the micro level (implement it with individually reversible operations).

3) Conservative logic: Conservative logic, introduced by Fredkin and Toffoli [68], [69], is a reversible computing model based on boolean circuits. Conservative logic circuits are composed of just two primitives - the *unit wire* and the *Fredkin gate*. A unit wire has a direction and transfers a signal from one of its ends to another in a single unit of time; it

does not support fan-out. The Fredkin gate has three inputs: a control signal c and two data lines. There are two output lines. If the control line c is 1, the output lines correspond exactly to the two data lines; if the control line is 0, the data lines are swapped. Note that the boolean function of the Fredkin gate is invertible and is its own inverse. Therefore a combinatorial circuit composed of Fredkin gates and unit wires can be reversed just by changing the direction of each unit wire. Moreover, the Fredkin gate always produces as many 1's in the output as were present in the input. Since the unit wire does not allow fan-out, conservative circuits always output as many 1's as were input (whence their name). Fredkin and Toffoli showed that conservative logic circuits can embed the computation of arbitrary boolean circuits, using an approach similar to Bennett's simulation of standard Turing machines by reversible Turing machines [68], [69].

Fredkin and Toffoli also described a physical manifestation of their conservative logic circuits, demonstrating that zero-dissipation reversible computation can theoretically be implemented with purely reversible physical forces. Billiard ball computers [68] (also known as "ballistic computers" [67]) model signals traveling through a conservative circuit as billiard balls. Collisions between balls and other objects are perceived to be perfectly elastic, changing a ball's direction by exactly 90°. The result of two balls colliding is also a perfect 90° rotation of the trajectory of each ball. A billiard ball "circuit" has input slots on one side, through which billiard balls are shot simultaneously with equal speed. The presence of a ball is equivalent to an input of 1 while the absence is a 0. The Fredkin gate can be implemented as a billiard ball circuit [68], which shows the equivalence between the billiard ball model and conservative circuits.

4) Space-time tradeoffs in computing reversibly: An irreversible computing process can always be simulated by a reversible process, but there are overheads. Bennett showed how to simulate an irreversible computation reversibly in polynomial time and at most quadratic space [70]. Lange, McKenzie, and Tapp later showed that it is possible to simulate an irreversible computation without using asymptotically more space; however, the run time of the simulation becomes exponential [71]. Further investigation has shown that these results are the two ends of a spectrum of time-space tradeoffs when simulating irreversible computation reversibly; an overview is available in [72].

E. Hardware for reverse computation

There has been some work in hardware support for enabling rollback of non-reversible programs; examples include Fujimoto, Tsai and Gopalakrishnan's rollback chip for optimistic simulation [73] and Doudalis and Prvuloic's hardware assisted reverse execution for debugging [21]. There have also been (very few) efforts to build a general-purpose computer that is fully reversible. Early efforts include Ressler's reversible computer based on conservative logic [74] and Hall's description of a reversible instruction set architecture [75]. The most complete effort to date is the Pendulum project, which included the design and fabrication of a fully reversible RISC-based processor and the associated memory [76], [77], [78], [79].

1) The Pendulum Instruction Set Architecture: The Pendulum instruction set is an instruction set for a Von Neumann computer that runs fully reversibly [77], [79]. Full reversibility requires both reversible data updates and reversible control flow constructs. Axelsen, Gück, and Yokoyama have formalized and generalized the Pendulum instruction set architecture and have proven its reversibility [80].

All data updates in the Pendulum instruction set are reversible. Pendulum has two types of register operations: non-expanding operations and expanding operations. Non-expanding register operations have the form $R_1 \leftarrow f(R_1, R_2)$; the old value of R_1 must be recoverable from the new value and R_2. Examples off non-expanding operations include `add` and `subtract`. Expanding operations, on the other hand, store their result in a third register because they are not reversible. For example, after the update $R_1 \leftarrow R_1$ `AND` R_2, the previous value of R_1 is no longer recoverable. Expanding operations have the form $R_3 \leftarrow R_3$ `XOR` $f(R_1, R_2)$; by using bitwise exclusive-or, they also preserve the value of the target register. All access to external memory occurs through `exchange` operations, rather than through reads and writes. An `exchange` swaps a word from memory with a value in a local register.

The branching instructions in Pendulum have a special structure to ensure reversibility. Each branching instruction has two parameters: a register to test against and a register containing the target address. The target address must point to an identical branching instruction. There is a global bit in the processor, named the branch bit, that is true immediately after a jump and false for sequential execution. If the branch bit is false and a branch instruction is reached, the branch condition is evaluated. If the branch condition is true, the program counter is swapped with the target address register, essentially performing a jump. When a branch instruction is entered with the branch bit set to true, it has been the target of a jump; it sets the branch bit to false and sequential execution continues.

III. Implementing incremental regenerative inversion

Both the extract-from-use techniques and redefine techniques [2] for regenerating previous values were implemented in Backstroke. Regeneration of previous values was implemented as a part of Backstroke's pluggable architecture; the extract-from-use technique and the redefine techniques were implemented as separate plugins that could be turned on and off independently. For example, for floating point values only the redefine technique is valid, while the extract-from-use technique is not.

After incremental regenerative inversion was implemented for loop-free code, we considered loops. Unfortunately, we realized that the definition-use analysis available at our disposal was not powerful enough to ensure correctness for code with

loops. In the presence of a loop, two uses of a variable may have the same set of reaching definitions, but the at runtime the variable may have different values at the two use sites. We realized that we needed an analysis that would partition variables values in such a way that we could guarantee that variables uses in the same partition refer to the same value. For this purpose we chose static single assignment.

IV. Source-level static single assignment analysis

A. The static single assignment analysis

Static single assignment (SSA) [81] is a classic intermediate form representation that simplifies a number of analyses and optimizations. In SSA form, each variable is replaced by a set of variables such that each variable is defined exactly once and each use has exactly one reaching definition.

In its standard formulation, SSA is an intermediate representation, used before code generation. Unfortunately, the intermediate representation formulation of SSA is not well-suited to source-to-source compilers. Modern developer toolchains often modify and analyze the source code for purposes other than compiling, such as refactoring. For example, the Clang [1] frontend for C and C++ is specifically designed to enable source-to-source compilers [2]. Out contribution is implementing SSA as an *analysis* rather than as an intermediate representation; the SSA information is attached to the original abstract syntax tree (AST).

SSA is defined in terms of scalar variables; when a structure is present it is treated as a set of the scalar variables that it contains. Modern object-oriented paradigms often involve deeply-nested structures and expanding such structures to their constituent scalar values could be computationally expensive. Moreover, treating structures as sets of scalar values fundamentally obscures the hierarchical relationships between these values. For example, take a structure `Line` that contains two `Point` objects, where each `Point` object holds three scalar values. A user of the SSA analysis might be interested at all the locations at which a `Line` object is modified, but is instead forced to query for definitions of its constituent scalar values. We implement a novel *hierarchical versioning* extension to standard SSA version numbers, so that any level of a structure can be queried independently.

In order to maintain the invariant that each use has exactly one reaching definition, SSA inserts ϕ functions at locations where two or more different definitions reach. Conceptually, a ϕ function has each reaching definition as a parameter and returns the correct definition based on the path taken dynamically. As a useful extension, we propose annotating each reaching definition to a ϕ function with a predicate that specifies the necessary and sufficient runtime conditions for selecting that definition in the ϕ function. This is useful in implementing the extract-from-use and redefine techniques for value restoration, as described in [2].

[1]http://clang.llvm.org/

[2]http://clang.llvm.org/comparison.html

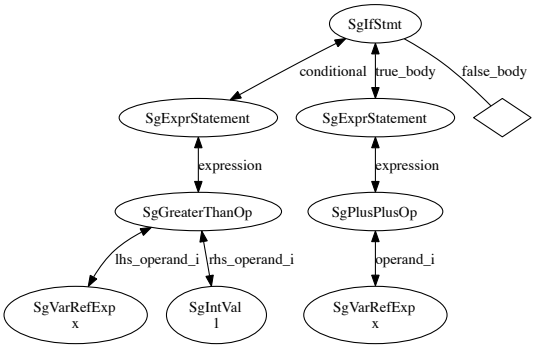

Fig. 4: A sample ROSE AST fragment for the source code `if (x > 3) { x++; }`

B. Algorithm Overview

The algorithm annotates the abstract syntax tree tree produced by the ROSE source-to-source compiler [3]. After dataflow is complete, each AST node is associated with a (variable name $\rightarrow$ definition) mapping representing the reaching definitions (defs) at that node. Since dataflow information is propagated along the control flow graph (CFG), rather than the AST, a one-to-one mapping between the control flow graph and the AST is imperative. In the full ROSE control flow graph, each AST node appears multiple times - once before the construct begins executing, once after the construct has finished executing, and once every time the node is visited during execution (e.g. loop). Figure 4 illustrates the AST for the code fragment `if (x > 3) { x++; }` and figure 5 shows the corresponding full CFG. In order to create a one-to-one mapping between AST nodes and CFG nodes, the CFG is filtered until each AST node appears exactly once. This filtering must be done carefully to ensure that reaching definitions are correctly propagated to each AST node. For example, in figure 5, we only include the last occurrence of `SgGreaterThanOp` because at runtime it executes after its arguments; meanwhile, we only kept the first occurrence of `SgIfStmt`. The implementation of correct data-flow for short-circuit evaluations (`&&` and `||`) remains a challenge, because the same AST node is in the CFG decision point an in one of the branch targets. The current implementation treats all logical operators as if both of their operands are evaluated.

The algorithm proceeds in five (mostly independent) main stages:

1) Discovery of variable names
2) Generation of local definition and use information
3) Inserting ϕ functions
4) Dataflow propagation of reaching definitions
5) Associating uses with reaching definitions

Stage 1 discovers all variables used in the program (including compound variables such as `x.a.b`) and generates unique

[3]http://www.rosecompiler.org

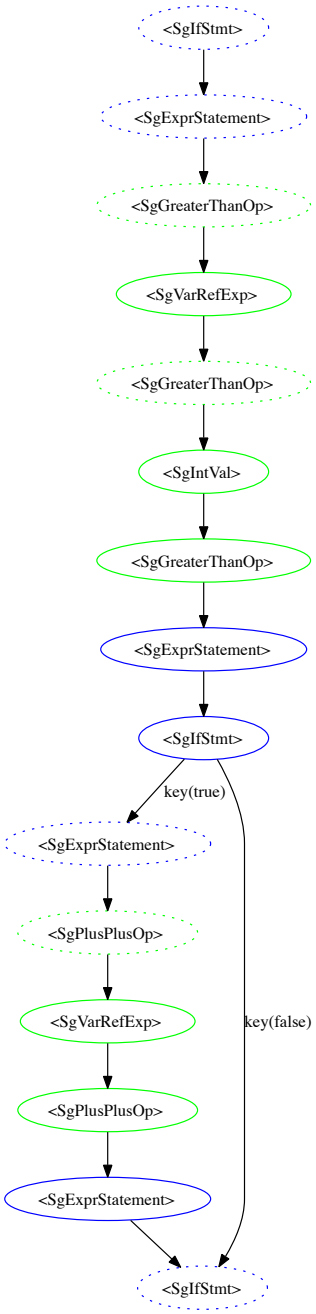

Fig. 5: The unfiltered ROSE control flow graph for the code fragment `if (x > 3) { x++; }` . The nodes that are filtered from the dataflow computation are shown with dotted borders.

names for them. All subsequent stages generate definition (def) and use information based on the variable names discovered in stage 1. Stage 2 collects the local defs and uses present at each AST node. Stage 2 also handles hierarchical versioning of the variables. Stage 3 inserts and numbers ϕ function according to the algorithm described in [81]. Finally, stage 4 propagates reaching definitions along the CFG in standard dataflow fashion. Stage 5 builds a table matching uses to reaching defs, to facilitate later queries. Stages 4 - 5 use standard dataflow techniques.

V. Implementing automatic checkpointing of C++ functions

Backstroke's checkpointing plugin statically determines what variables are potentially modified by the event method and saves those variables. The reverse method restores the values of the saved variables, without executing the original method in reverse. There are times when reverse execution is more efficient, while other times state saving is more efficient. The Backstroke architecture allows a different reverse code generation plugin to be used for each nested scope, thus it an optimal inversion can apply the checkpointing plugin while using reverse execution where it would be appropriate and more efficient.

A. Interprocedural checkpointing algorithm

The Backstroke checkpointing plugin uses interprocedural dataflow analysis [82] to determine which variables are potentially modified by the event method. First, a call graph is constructed and all functions reachable from the event method are analyzed for local definitions and uses. Then, definitions are propagated up in the call graph; if a callee modifies a variable, so does the caller. If function arguments are passed by reference or pointer, the formal argument in the callee is aliased with the actual argument in the caller. For member functions, the implicit `this` argument is always aliased with the object instance in the caller. Recursion is handled by iterating over the call graph until no new definitions are propagated to any function.

The Backstroke checkpointing plugin uses a very generic software analysis to determine which variables are potentially modified; as a consequence it could sometimes save variables that are not part of the simulation state. For instance, data structures associated with the simulation engine (such as the pending event set) should not be explicitly modified by the reverse methods. It might also be preferable to ignore the state of performance counters, debugging logs, etc. To address this, Backstroke allows for an optional variable filter to be provided. The variable filter is a function that takes in a variable as a parameter and returns `true` is the variable should be restored during rollback, and `false` otherwise. In future versions we plan to include support for `#pragma` declarations that the practitioner can use to mark variables that are not part of the simulation state.

Once the Backstroke checkpointing plugin prepares a list of the variables modified by an event method, it generates the

code to save and restore these variables. In the very beginning of the forward method, all modified variables are pushed onto a deque (double-ended queue) data structure. The rest of the forward method is identical to the original event method. The reverse method pops the saved values from the back of the deque and restores the modified variables to their old values. In contrast, the commit method removes the saved values from the front of the deque and discards them. The deque is suitable storage because reverse methods are always guaranteed to be called in reverse order of the forward methods, starting with the last forward method called. Meanwhile, commit methods are guaranteed to be called in the same order as the forward methods, starting with the first forward method called.

B. Limitations

Due to the complexity of the C++ language, not all language constructs can be safely handled by the state saving plugin. The language checking phase of Backstroke scans the bodies of all functions reachable from the body of an event method and issues errors when unsafe constructs are encountered. Pointer arithmetic is not supported, since it makes it impossible to statically determine the variables that are being modified. Although the use of pointers is supported, Backstroke currently does not have aliasing analysis; as a result if the same variable has multiple aliases it may be stored multiple times. State-saving of dynamically allocated arrays is not supported, since their size cannot be statically determined. However, when arrays are encapsulated inside a data structure such as `std::vector`, they can be saved and restored successfully. Static variables declared in function scope are another problematic C/C++ construct. Such variables form part of the persistent program state, but are solely accessible from a single function. Perhaps in the future such static variables can be handled by generating a globally unique name and automatically hoisting their declaration to global scope.

Saving and restoring the values of C++ classes is also potentially dangerous since doing so necessarily invokes the object's copy constructor. If the copy constructor has side effects on the global simulation state, the act of saving the simulation state may itself modify it. Luckily, this is not a major restriction because having copy constructors with global side effects is a particularly bad programming practice! A similar warning applies to overloaded assignment operators, since such operators are used by the reverse method to restore the value of class objects. A more subtle issue with saving class objects are classes whose copy constructors do not create deep copies. For example, copying a smart pointer type such as `std::auto_ptr` creates a new object, but the object references the same underlying data. It is not possible to statically verify that a class's copy constructor correctly creates a deep copy of the class; consequently Backstroke issues a warning when relying on the constructor semantics of classes that contain non-const pointers. Backstroke also includes explicit support for standard smart pointer types, such as `std::auto_ptr` and `boost::shared_ptr`[4].

VI. Experimental study: Applying Backstroke to automatically produce optimistic HLA federations

We postulated that using Backstroke, it would be possible to take a sequential simulation and self-federate it using HLA [83] automatically obtain an optimistic distributed simulation. To validate the approach, we applied the proposed methodology to a small queueing network simulation. The simulation in question was a gasoline station simulation motivated by that described in [84]. Although not a large model, it is representative of the queueing models typically modeled by discrete event simulation. We intentionally took a model written by a person unacquainted with Backstroke, so that it could represent a realistic usage of C++ language constructs. The gasoline station event handlers were member functions and used templates, namespaces, C++ standard library data structures, and Boost smart pointers [4]. Furthermore, the event handling methods used utility functions to poll and advance the queues, hence testing Backstroke's interprocedural analysis.

The gasoline station instances were federated in an arbitrary topology using the HLA data management services. Each consumer leaving a gas station was randomly routed to one of the subsequent stations according to predefined probabilities. Certain gas station instances were designated as sources, while others acted as sinks for a randomly selected portion of consumers. The travel times between the gas stations were drawn from an exponential distribution. Although an exponential distribution is physically unrealistic for vehicles, we chose it because it is an example of a model with little inherent lookahead. Synchronizing such a federation conservatively would require advanced lookahead extraction techniques, such modifying the implementation to presample service times [85].

The Time Warp local control mechanism was inserted by extending the sequential simulation engine with HLA primitives. There was no naming convention for the event handling methods, so the event detection was performed by supplying a list of the names of all the event handling methods in the simulation. To prevent Backstroke from saving and restoring the discrete event kernel object, we filtered all objects with type `DESEngine` from the simulation state (as described in section V-A). As a final step, the simulation was built using the Backstroke tool instead of `g++`, producing the forward, reverse, and commit methods.

The federated gasoline station instances ran together optimistically with no significant issues. The execution exercised rollback, message retraction, and fossil collection and the results were repeatable. Although the example used in this experiment is a modest sized simulation, this experiment did serve our goal of exercising our approach to automating the generation of optimistic federated simulation code.

[4] http://www.boost.org/doc/libs/release/libs/smart_ptr/smart_ptr.htm

VII. Applying Backstroke to the Georgia Tech Network Simulator

GTNetS consists of over 200,000 lines of real-world C++ code and simulates all layers of the network stack [5]. It is an example of discrete event simulator that is not designed for distributed execution or automatic inversion, and thus includes usage of the full range of C++ features. Some of the problematic C++ aspects of GTNetS include:

- Extensive aliasing.
- Function pointers.
- Pointer arithmetic.
- Deep inheritance hierarchies with many virtual methods.
- Large objects without valid copy constructors or destructors.
- Use of untyped data, such as `void*` pointers and `reinterpret_cast`.
- An almost fully-connected memory graph; the whole simulation state was accessible through any simulation object
- Hidden state; part of the simulation state was not directly reachable from the event methods. Examples of hidden state include static variables declared in function scope.
- Constructors and destructors that modify the simulation's state; creating a copy of an object with such a constructor could cause the simulation state to be modified.

We chose a wireless network simulation built on top of GTNetS and tested two of Backstroke's inversion methods on it: regenerative inversion and checkpointing. Due to the complexity of the C++ constructs used in GTNetS, neither method performed well. Regenerative inversion successfully reproduced the original simulation trace, even after each event was rolled back once and rexecuted. However, the state saving overhead was very large due to the connectivity of the GTNetS memory graph and its large objects. Backstroke's implementation of checkpointing was thwarted by GTNetS's hidden state and its extensive use of virtual methods and inheritance, which made interprocedural analysis difficult. Checkpointing was not able to reproduce the correct simulation results after a rollback.

Although our initial effort failed to produce correct reverse methods for GTNetS, the study exposed important weaknesses of the inversion methods implemented in Backstroke at the time. We reevaluated our approach to inverting complex C++ constructs and found efficient ways to invert all the problematic constructs listed earlier; our findings are set to appear in the 2011 Winter Simulation Conference [86]

References

[1] T. Akgul and V. J. Mooney III, "Instruction-level reverse execution for debugging," in *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '02*, vol. 28, no. 1. ACM, Jan. 2002, pp. 18–25. [Online]. Available: http://dx.doi.org/10.1145/634636.586101

[2] ——, "Assembly instruction level reverse execution for debugging," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 2, pp. 149–198, Apr. 2004. [Online]. Available: http://dx.doi.org/10.1145/1018210.1018211

[3] T. Akgul, "Assembly instruction level reverse execution for debugging," Ph.D. dissertation, Georgia Institute of Technology, Apr. 2004. [Online]. Available: http://dx.doi.org/1853/5249

[4] G. F. Riley, M. Ammar, R. M. Fujimoto, A. Park, K. S. Perumalla, and D. Xu, "A federated approach to distributed network simulation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 14, no. 2, pp. 116–148, Apr. 2004. [Online]. Available: http://dx.doi.org/10.1145/985793.985795

[5] G. F. Riley, "The Georgia Tech Network Simulator," in *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, no. August. ACM, 2003, pp. 5–12. [Online]. Available: http://dx.doi.org/10.1145/944773.944775

[6] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, p. 425, 1985. [Online]. Available: http://dx.doi.org/10.1145/3916.3988

[7] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 9, no. 3, pp. 224–253, 1999. [Online]. Available: http://dx.doi.org/10.1145/347823.347828

[8] Y. Tang, K. S. Perumalla, and R. M. Fujimoto, "Optimistic simulations of physical systems using reverse computation," *Simulation*, 2006. [Online]. Available: http://dx.doi.org/10.1177/0037549706065481

[9] R. M. Fujimoto and A. Naborskyy, "Using Reversible Computation Techniques in a Parallel Optimistic Simulation of a Multi-Processor Computing System," *21st International Workshop on Principles of Advanced and Distributed Simulation (PADS'07)*, pp. 179–188, Jun. 2007. [Online]. Available: http://dx.doi.org/10.1109/PADS.2007.31

[10] C. Carothers and M. Peters, "An algorithm for fully-reversible optimistic parallel simulation," in *Simulation Conference, 2003. Proceedings of the 2003*, 2003. [Online]. Available: http://dx.doi.org/10.1109/WSC.2003.1261505

[11] D. W. Bauer and E. H. Page, "An Approach for Incorporating Rollback through Perfectly Reversible Computation in a Stream Simulator," in *21st International Workshop on Principles of Advanced and Distributed Simulation (PADS'07)*. IEEE Computer Society, Jun. 2007, pp. 171–178. [Online]. Available: http://dx.doi.org/10.1109/PADS.2007.13

[12] H. Baker, "NREVERSAL of fortune-the thermodynamics of garbage collection," in *Proceedings of the International Workshop on Memory Management*, vol. 91436, no. 818. Springer, 1992, pp. 507–524. [Online]. Available: http://dx.doi.org/10.1007/BFb0017210

[13] R. Balzer, "EXDAMS - EXtendable Debugging and Monitoring System," in *AFIPS '69 (Spring): Proceedings of the May 14-16, 1969, spring joint computer conference*. Boston, Massachusetts: ACM, 1969, pp. 567–580. [Online]. Available: http://dx.doi.org/10.1145/1476793.1476881

[14] M. V. Zelkowitz, "Reversible execution," *Communications of the ACM*, vol. 16, no. 9, p. 566, Sep. 1973. [Online]. Available: http://dx.doi.org/10.1145/362342.362360

[15] S. Feldman and C. Brown, "Igor: A system for program debugging via reversible execution," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*. ACM, 1988, p. 123. [Online]. Available: http://dx.doi.org/10.1145/69215.69226

[16] H. Agrawal, R. DeMillo, and E. Spafford, "An execution backtracking approach to program debugging," *IEEE Software*, vol. 8, no. 3, pp. 21–26, 1991. [Online]. Available: http://dx.doi.org/10.1109/52.88940

[17] R. Netzer and M. Weaver, "Optimal tracing and incremental reexecution for debugging long-running programs," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM New York, NY, USA, 1994, pp. 313–325. [Online]. Available: http://dx.doi.org/10.1145/178243.178477

[18] S. Booth and S. Jones, "Walk backwards to happiness: debugging by time travel," *AADEBUG'97*, no. July, p. 171, 1997. [Online]. Available: http://www.ep.liu.se/ea/cis/1997/009/14/

[19] B. Biswas and R. Mall, "Reverse execution of programs," *ACM SIGPLAN Notices*, vol. 34, no. 4, pp. 61–69, Apr. 1999. [Online]. Available: http://dx.doi.org/10.1145/312009.312079

[20] S. Chen, W. Fuchs, and J. Chung, "Reversible debugging using program instrumentation," *IEEE transactions on software*, vol. 27, no. 8, pp. 715–728, 2001. [Online]. Available: http://dx.doi.org/10.1109/32.940726

[21] I. Doudalis and M. Prvulovic, "HARE: Hardware assisted reverse execution," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, Jan. 2010, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2010.5416651

[22] R. Sauciuc and G. Necula, "Reverse Execution With Constraint Solving," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-67, May 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-67.html

[23] J. Archer, J.E., R. Conway, and F. Schneider, "User recovery and reversal in interactive systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 6, no. 1, pp. 1–19, Jan. 1984. [Online]. Available: http://dx.doi.org/10.1145/357233.357234

[24] J. S. Briggs, "Generating reversible programs," *Software: Practice and Experience*, vol. 17, no. 7, pp. 439–453, Jul. 1987. [Online]. Available: http://dx.doi.org/10.1002/spe.4380170703

[25] P. Bishop, "Using reversible computing to achieve fail-safety," in *Proceedings The Eighth International Symposium on Software Reliability Engineering*. IEEE Computer Society, 1997, pp. 182–191. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.1997.630863

[26] R. S. Bird, "An introduction to the theory of lists," in *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*. New York, NY, USA: Springer-Verlag New York, Inc., 1987, pp. 5–42.

[27] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi, "Automatic inversion generates divide-and-conquer parallel programs," in *ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*. New York, New York, USA: ACM Press, 2007, p. 146. [Online]. Available: http://dx.doi.org/10.1145/1250734.1250752

[28] R. J. Ross, "Experience with the DYNAMOD Program Animator," in *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, vol. 23, no. 1. ACM, 1991, pp. 35–42. [Online]. Available: http://dx.doi.org/10.1145/107005.107013

[29] M. R. Birch, C. M. Boroni, F. W. Goosey, S. D. Patton, D. K. Poole, C. M. Pratt, and R. J. Ross, "DYNALAB: a dynamic computer science laboratory infrastructure featuring program animation," in *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education - SIGCSE '95*, vol. 27, no. 1. ACM, 1995, pp. 29–33. [Online]. Available: http://dx.doi.org/10.1145/199688.199706

[30] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi, "Reversible Execution and Visualization of Programs with LEONARDO," *Journal of Visual Languages & Computing*, vol. 11, no. 2, pp. 125–150, Apr. 2000. [Online]. Available: http://dx.doi.org/10.1006/jvlc.1999.0143

[31] Z. Hu, S. Mu, and M. Takeichi, "An algebraic approach to bi-directional updating," *Programming Languages and Systems*, vol. 3302, pp. 2–20, 2004. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30477-7_2

[32] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bi-directional tree transformations," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages - POPL '05*. New York, New York, USA: ACM Press, 2005, pp. 233–246. [Online]. Available: http://dx.doi.org/10.1145/1040305.1040325

[33] J. N. Foster, "Bidirectional programming languages," Ph.D. dissertation, University of Pennsylvania, 2010. [Online]. Available: http://repository.upenn.edu/cis_reports/921/

[34] S. W. Golomb and L. D. Baumert, "Backtrack Programming," *Journal of the ACM*, vol. 12, no. 4, pp. 516–524, Oct. 1965. [Online]. Available: http://dx.doi.org/10.1145/321296.321300

[35] R. W. Floyd, "Nondeterministic Algorithms," *Journal of the ACM*, vol. 14, no. 4, pp. 636–644, Oct. 1967. [Online]. Available: http://dx.doi.org/10.1145/321420.321422

[36] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Society for Industrial and Applied Mathematics (SIAM), 2008.

[37] A. Griewank, D. Juedes, and J. Utke, "Algorithm 755; ADOL-C: a package for the automatic differentiation of algorithms written in C/C++," *ACM Transactions on Mathematical Software*, vol. 22, no. 2, pp. 131–167, Jun. 1996. [Online]. Available: http://dx.doi.org/10.1145/229473.229474

[38] J. Grimm, L. Pottier, and N. Rostaing-Schmidt, "Optimal time and minimum space-time product for reversing a certain class of programs," Institut National de Recherche en Informatique et en Automatique, Tech. Rep., 1996. [Online]. Available: http://dx.doi.org/10068/40845

[39] A. Walther, "Program reversal schedules for single- and multi-processor machines," Ph.D. dissertation, Technical University Dresden, 1999. [Online]. Available: http://deposit.ddb.de/cgi-bin/dokserv?idn=96395007x&dok_var=d1&dok_ext=pdf&filename=96395007x.pdf

[40] U. Naumann, "On optimal DAG reversal," RWTH Aachen, Tech. Rep. March, 2009. [Online]. Available: http://ftp.informatik.rwth-aachen.de/ftp/pub/publications/rwth/informatik/2007/2007-05.pdf

[41] J. McCarthy, "The Inversion of Functions Defined by Turing Machines," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, 1956, pp. 177–181.

[42] E. W. Dijkstra, "Program Inversion," in *Program Construction, International Summer School*. London, UK: Springer-Verlag, 1979, pp. 54–57.

[43] D. Gries, "Inverting Programs," in *The science of programming*. New York, New York, USA: Springer-Verlag, 1981, ch. 21, pp. 265–274.

[44] C. Lutz and H. Derby, "JANUS : A TIME-REVERSIBLE LANGUAGE," 1982. [Online]. Available: http://web.archive.org/web/20070212152136/http://www.cise.ufl.edu/~mpf/rc/janus.html

[45] T. Yokoyama and R. Glück, "A reversible programming language and its invertible self-interpreter," in *ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '07*. New York, New York, USA: ACM Press, 2007, p. 144. [Online]. Available: http://dx.doi.org/10.1145/1244381.1244404

[46] W. Chen and J. Udding, "Program inversion: More than fun!" *Science of Computer Programming*, vol. 15, no. 1, pp. 1–13, Nov. 1990. [Online]. Available: http://dx.doi.org/10.1016/0167-6423(90)90042-C

[47] D. Eppstein, "A heuristic approach to program inversion," in *Int. Joint Conference on Artificial Intelligence (IJCAI-85)*. Citeseer, 1985, pp. 219–221. [Online]. Available: http://portal.acm.org/citation.cfm?id=1625175

[48] R. Korf, "Inversion of applicative programs," in *Proceedings of the Seventh Intern. Joint Conference on Artificial Intelligence (IJCAI-81)*, no. 3597, 1981, pp. 1007–1009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1623345

[49] R. Glück and M. Kawabe, "Revisiting an automatic program inverter for Lisp," *ACM SIGPLAN Notices*, vol. 40, no. 5, pp. 8–17, May 2005. [Online]. Available: http://dx.doi.org/10.1145/1071221.1071222

[50] M. Kawabe and R. Glück, "The program inverter LRinv and its structure," *Practical Aspects of Declarative Languages*, pp. 219–234, 2005. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30557-6_17

[51] R. Glück and M. Kawabe, "Derivation of deterministic inverse programs based on LR parsing," *Functional and Logic Programming*, vol. 2988, pp. 187–191, 2004. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24754-8_21

[52] R. Glück and S. Abramov, "Principles of inverse computation and the universal resolving algorithm," *The Essence of Computation*, vol. 2566, pp. 269–295, 2002. [Online]. Available: http://dx.doi.org/10.1007/3-540-36377-7_13

[53] S. Abramov and R. Glück, "The Universal Resolving Algorithm: Inverse Computation in a Functional Language," *Science of Computer Programming*, vol. 43, no. 2-3, pp. 193–229, 2002. [Online]. Available: http://dx.doi.org/10.1016/S0167-6423(02)00023-0

[54] Y. Futamura, "Partial Evaluation of Computation ProcessâĂŤ An Approach to a Compiler-Compiler," *Systems, Computers, Controls*, vol. 2, no. 5, pp. 45–50, 1971. [Online]. Available: http://dx.doi.org/10.1023/A:1010095604496

[55] R. Glück, Y. Kawada, and T. Hashimoto, "Transforming interpreters into inverse interpreters by partial evaluation," in *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. ACM, 2003, pp. 10–19. [Online]. Available: http://dx.doi.org/10.1145/777388.777391

[56] B. J. Ross, "Running programs backwards: The logical inversion of imperative computation," *Formal Aspects of Computing*, vol. 9, no. 3, pp. 331–348, May 1997. [Online]. Available: http://dx.doi.org/10.1007/BF01211087

[57] J. Lee, "A Case for Dynamic Reverse-code Generation to Debug Non-deterministic Programs," BRICS Research Series, Tech. Rep., 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.6400http://formal.korea.ac.kr/~jlee/papers/rcg-case.pdfhttp://people.cis.ksu.edu/~jlee/papers/rcg-case.pdf

[58] ——, "Dynamic Reverse Code Generation for Backward Execution," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 4, pp. 37–54, May 2007. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2006.12.028

[59] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, "Path-based inductive synthesis for program inversion," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and*

implementation - PLDI '11. New York, New York, USA: ACM Press, 2011, p. 492. [Online]. Available: http://dx.doi.org/10.1145/1993498.1993557

[60] T. Yokoyama, H. B. Axelsen, and R. Glück, "Principles of a reversible programming language," in *Proceedings of the 2008 conference on Computing frontiers - CF '08*. New York, New York, USA: ACM Press, 2008, p. 43. [Online]. Available: http://dx.doi.org/10.1145/1366230.1366239

[61] T. Mogensen, "Partial evaluation of the reversible language janus," in *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM, 2011, pp. 23–32. [Online]. Available: http://dx.doi.org/10.1145/1929501.1929506

[62] M. Frank, "The R Programming Language and Compiler," MIT AI Lab, Tech. Rep., 1997. [Online]. Available: http://web.archive.org/web/20041010220454/http://www.cise.ufl.edu/~mpf/rc/memos/M08/M08_rdoc.html

[63] Z. Hu, S. Mu, and M. Takeichi, "An injective language for reversible computation," *Mathematics of Program Construction*, vol. 3125, pp. 289–313, 2004. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27764-4_16

[64] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM Journal of Research and Development*, vol. 44, no. 1, pp. 261–269, 1961. [Online]. Available: http://dx.doi.org/10.1147/rd.53.0183

[65] M. Frank, "Introduction to reversible computing: motivation, progress, and challenges," in *Proceedings of the 2nd Conference on Computing Frontiers*. ACM New York, NY, USA, 2005, pp. 385–390. [Online]. Available: http://dx.doi.org/10.1145/1062261.1062324

[66] C. Bennett, "Logical Reversibility of Computation," *IBM journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973. [Online]. Available: http://dx.doi.org/10.1147/rd.176.0525

[67] ——, "The thermodynamics of computation - a review," *International Journal of Theoretical Physics*, vol. 21, no. 12, pp. 905–940, 1982. [Online]. Available: http://dx.doi.org/10.1007/BF02084158

[68] E. Fredkin and T. Toffoli, "Conservative logic," *International Journal of Theoretical Physics*, vol. 21, no. 3-4, pp. 219–253, Apr. 1982. [Online]. Available: http://dx.doi.org/10.1007/BF01857727

[69] T. Toffoli, "Reversible Computing," MIT Laboratory for Computer Science, Tech. Rep., 1980. [Online]. Available: http://dx.doi.org/10.1007/3-540-10003-2_104

[70] C. Bennett, "Time/space trade-offs for reversible computation," *SIAM Journal on Computing*, vol. 18, no. 4, p. 766, 1989. [Online]. Available: http://dx.doi.org/10.1137/0218053

[71] K. Lange, P. McKenzie, and A. Tapp, "Reversible space equals deterministic space," in *Computational Complexity, 1997. Proceedings., Twelfth Annual IEEE Conference on (Formerly: Structure in Complexity Theory Conference)*. IEEE Comput. Soc, 1997, pp. 45–50. [Online]. Available: http://dx.doi.org/10.1109/CCC.1997.612299

[72] P. Vitányi, "Time, space, and energy in reversible computing," *Proceedings of the 2nd conference on Computing*, pp. 435–444, 2005. [Online]. Available: http://dx.doi.org/10.1145/1062261.1062335

[73] R. M. Fujimoto, J.-J. Tsai, and G. Gopalakrishnan, "Design and evaluation of the rollback chip: special purpose hardware for Time Warp," *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 68–82, 1992. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=123382

[74] A. Ressler, "The Design of a Conservative Logic Computer and a Graphical Editor Simulator," Masters Thesis, 1981. [Online]. Available: http://dx.doi.org/1721.1/15895

[75] J. S. Hall, "A reversible instruction set architecture and algorithms," in *Physics and Computation, 1994. PhysComp'94, Proceedings., Workshop on*. IEEE, 1994, pp. 128–134. [Online]. Available: http://dx.doi.org/10.1109/PHYCMP.1994.363690

[76] C. Vieri, "Reversible computer engineering and architecture," Ph.D. dissertation, Massachusetts Institute of Technology, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=930761

[77] ——, "Pendulum: A reversible computer architecture," Masters thesis, Massachusetts Institute of Technology, 1995. [Online]. Available: http://dx.doi.org/1721.1/36039

[78] M. Frank, "Reversibility for efficient computing," Ph.D. dissertation, 1999. [Online]. Available: http://dx.doi.org/1721.1/9464

[79] C. Vieri, M. Ammer, M. Frank, N. Margolus, and T. Knight, "A Fully Reversible Asymptotically Zero Energy Microprocessor," in *Power Driven Microarchitecture Workshop*. Citeseer, 1998. [Online]. Available: http://dx.doi.org/10.1109/PHYCMP.1994.363690

[80] H. Axelsen, R. Glück, and T. Yokoyama, "Reversible Machine Code and Its Abstract Processor Architecture," *Computer ScienceâĂŞTheory and Applications*, pp. 56–69, 2007. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74510-5_9

[81] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://dx.doi.org/10.1145/115372.115320

[82] A. Aho, R. Sethi, and J. Ullman, "Compilers: principles, techniques, and tools." Reading, MA: Addison-Wesley Publishing Co., 1986, pp. 660–680.

[83] J. S. Dahmann, R. M. Fujimoto, and R. Weatherly, "The Department of Defense High Level Arhchitecture," in *Proceedings of the 29th conference on Winter simulation*. IEEE Computer Society, 1997, pp. 142–149. [Online]. Available: http://dx.doi.org/10.1145/268437.268465

[84] L. Birta and G. Arbez, *Modelling and Simulation*, 1st ed. Springer, 2007.

[85] D. M. Nicol, "Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks," in *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*. ACM, 1988, pp. 124–137. [Online]. Available: http://dx.doi.org/10.1145/62115.62128

[86] G. Vulov, C. Hou, D. Quinlan, R. Vuduc, R. M. Fujimoto, and D. R. Jefferson, "The Backstroke framework for source-level reverse computation applied to parallel discrete event simulation," in *Proceedings of the 2011 Winter Simulation Conference (to appear)*, 2011.