Proceedings of the 2011 Winter Simulation Conference
S. Jain, R.R. Creasey, J. Himmelspach, K.P. White, and M. Fu, eds.

THE BACKSTROKE FRAMEWORK FOR SOURCE LEVEL REVERSE COMPUTATION APPLIED TO PARALLEL DISCRETE EVENT SIMULATION

George Vulov
Cong Hou
Richard Vuduc
Richard Fujimoto

Georgia Institute of Technology
Atlanta, Georgia USA

Daniel Quinlan
David Jefferson

Lawrence Livermore National Laboratory
Livermore, California USA

ABSTRACT

We introduce Backstroke, a new open source framework for the automatic generation of reverse code for functions written in C++. Backstroke enables *reverse computation* for optimistic parallel discrete event simulations. It is built using the ROSE open-source compiler infrastructure, and handles complex C++ features including pointers and pointer types, arrays, function and method calls, class types, inheritance, polymorphism, virtual functions, abstract classes, templated classes and containers. Backstroke also introduces new program inversion techniques based on advanced compiler analysis tools built into ROSE. We explore and illustrate some of the complex language and semantic issues that arise in generating correct reverse code for C++ functions.

1 INTRODUCTION

Backstroke is a new open source framework for the automatic generation of reverse code for functions written in C++. It is built on ROSE, a widely used source-to-source compiler infrastructure that has been under development at Lawrence Livermore National Laboratory for a decade (Quinlan 2011). The primary purpose of Backstroke is to enable reverse computation for fast, efficient rollback in optimistic parallel discrete event simulations.

Reverse computation is a well established technique now, but major practical barriers still prevent its widespread adoption. It is difficult to create correct and efficient reverse methods by hand, and unrealistic to require programmers to do so. Further, since the number of times an event is rolled back is nondeterministic, simple bugs in reverse code may behave as "Heisenbugs" even though the reverse code itself is sequential and deterministic. This can make debugging very difficult. These considerations suggest *automatic* generation of reverse code (Perumalla 1999, Carothers, Perumalla, and Fujimoto 1999). Backstroke takes this approach, but advances it considerably beyond prior systems. In particular, it will reverse more complex language constructs than prior work, since it applies to C++ rather than just C, and it serves as a general framework allowing experimentation with many different inversion techniques. We will describe some of the general approaches to automatic generation of reverse code, and explore language and semantic issues that arise in generating correct reverse code for C++ functions.

2 CONTEXT AND RELATED WORK

A parallel discrete event simulation (PDES) program consists of *logical processes* (LPs) that execute concurrently and exchange *timestamped event messages*. A synchronization algorithm is required to ensure each LP processes events in non-decreasing timestamp order. *Optimistic* synchronization uses rollback to undo the computation of events that are performed out of timestamp order (Jefferson 1985). Parallel discrete event simulation and optimistic execution are described in detail in (Fujimoto 2000).

978-1-4577-2109-0/11/$26.00 ©2011 IEEE

Among the well-known optimistic parallel simulators are the Time Warp Operating System (Jefferson et al. 1987), SPEEDES (Steinman 1992), Georgia Tech Time Warp (Das et al. 1994), PARSEC (Bagrodia et al. 1998), ROSS (Carothers 2002), μsik (Perumalla 2005), and others (Subramania and Thazhuthaveetil 2001).

Reverse computation is a technique to efficiently implement event rollback (Carothers, Perumalla and Fujimoto, 1999; Tang Perumalla, and Fujimoto 2006, Naborskyy and Fujimoto 2007). Each event routine is instrumented to save a minimal amount of control and data information before and during the forward execution to allow later reconstruction of the state of the LP just prior to executing the event. The reconstruction uses algebraic combinations of the final values of state variables after the event and the values of the saved data.

Backstroke is not a single method or algorithm, but rather a *framework* designed to accommodate a collection of reverse computation or program inversion methods. Different inversion techniques can be used in different regions of the simulation code as appropriate, or can be applied for different variables of the simulation state depending on their types and usage. Over time we expect Backstroke to evolve by accumulating an ever larger, more sophisticated library of inversion techniques along with more precise criteria for deciding when to use each one.

Backstroke's reverse computation capability can be used in a number of ways to automate the creation of optimistic parallel discrete event simulations. It can be used for simulations developed *de novo*, but it can also help create an optimistic simulation by federating two or more sequential models (Nicol and Heidelberger 1995; Wang 2005; Riley et al. 2004), or help transform a conservative simulation into an optimistic one (Santoro and Quaglia 2006). In principle it could also be used for other reverse execution applications as well, such as program debugging (Zelkowitz 1973; Feldman and Brown 1988; Agrawal, Demillo, and Spafford 1991; Akgul and Mooney 2004), implementing *undo* in end-user applications (Archer et al. 1984; Briggs 1987), optimistic transaction synchronization (Jefferson and Motro 1986), and some kinds of fault recovery (Bishop 1997). We do not address these applications here.

Backstroke extends the range of language constructs to which reverse computation methods apply. Previous work in applying reverse computation to optimistic simulation has generally been limited to events written in C (Carothers et al. 1999). Backstroke, however, is designed to handle the full complexity of C++ including pointers and pointer types, function calls, class types, inheritance, polymorphism, virtual functions, function pointers, abstract classes, templated classes and containers, etc.

Even for code in the C-like subset of C++, Backstroke introduces new program inversion techniques that go well beyond those described in previous studies. Path-oriented inversion methods make use of sophisticated program analysis tools, e.g. SSA (static single assignment) value-graph analysis of the flow of data in a function, and interprocedural analysis. Such methods also can handle inversion of escape constructs such as `return`, `break`, `continue`, and `throw` that cannot be handled in full generality by previous techniques.

3 WHAT BACKSTROKE DOES

Backstroke takes all the source code files for a simulation written in C++ and produces modified source code in which all event methods in the simulation have three new associated methods that are declared and defined in the same scope as the original method. For each event method `E()` declared as

```
void E(args)
```

the Backstroke system generates three related methods, `E_forward(args)`, `E_reverse(args)`, and `E_commit()`. Roughly speaking these new methods do the following:

`void E_forward(args)` has essentially the same effect on the simulation state as `E(args)`, but in addition it saves data on a *rollback stack* (implemented as a double-ended queue) that may be needed in case of rollback to reconstruct the original state before `E(args)` was called.

`void E_reverse(args)` rolls back the side effects done by `E_forward(args)`, thereby restoring the LP to the state it was in before `E_forward(args)` was executed. In so doing it pops off the data saved on the rollback stack during the execution of `E_forward(args)`.

`void E_commit()` performs actions that are either irreversible or are unsafe to reverse if executed speculatively in `E_forward(args)`. They thus must be executed later at *commit time*, when it is certain that the event will not be rolled back. These actions include writing to output streams and deallocating dynamically allocated variables, but some other actions may need to be treated this way as well. `E_commit()` executes at an unspecified time after `E_forward()` and is constrained to have no access to the simulation state, so that it neither reads nor modifies it.

More formally, for a `void` event method `E(args)` the derived methods are required to satisfy the following semantic equations:

(a) `{ E_forward(args); E_commit(); }        ≡ E(args)`

(b) `{ E_forward(args); E_reverse(args); } ≡ {}`

Equation (a) says roughly that any event method `E(args)` is *equivalent*, in the sense of having indistinguishable side effects on simulation state and identical simulation output, to executing `E_forward(args)` followed by `E_commit(args)`. Thus, whenever in an optimistic simulation an event `E` is executed for the last time, so that only `E_forward(args)` and later `E_commit()` are executed, the behavior is the same as if the original event method `E(args)` was executed exactly once.

Equation (b) says that `E_forward(args)` followed by `E_reverse(args)` is equivalent to a no-op, i.e. that `E_reverse` exactly undoes all of the side-effects of `E_forward` (and does nothing else), leaving the computation in a state equivalent to having done nothing at all, so that `E_reverse` correctly accomplishes a rollback.

These equations are not sufficient to perfectly define the required behavior of the three derived methods. They do not, for example, capture the fact that `E_commit()` cannot access the LP state variables. Furthermore, method `E_forward()` followed by `E_commit()` is not required to leave the computation in *exactly* the same binary state, bit for bit, as executing `E()`, and `E_forward()` followed by `E_reverse()` also does not have to leave the computation perfectly unchanged in a binary sense. We also do require that `E_forward()` followed by `E_reverse()` produces no memory leaks. In both cases, however, the before- and after-LP states must be *equivalent* in a sense to be made more precise in Section 6.7. Despite these limitations these two equations are very useful in clarifying the requirements for reverse computation.

4 BACKSTROKE AS A ROSE APPLICATION

Backstroke is based on the ROSE source-to-source code compiler infrastructure (Quinlan 2011). ROSE is a compiler in that it inputs a source program and outputs an executable binary, but it also allows arbitrary code transformations within the front end. A single execution of ROSE involves the following steps, as illustrated in Figure 1:

- The source code for the entire program is parsed and converted to an Abstract Syntax Tree (AST), along with other analytically useful data structures.
- The AST is then modified or transformed according to the logic of the particular ROSE application. In the case of Backstroke this transformation is the generation of forward, reverse, and commit methods for every event method.
- After the transformation of the AST, ROSE then *unparses* it to produce C++ source code again.
- Finally, the modified code goes through a normal compile-and-build process to produce an executable binary.

The second step is the key part of the process that distinguishes one ROSE application from another. Backstroke, as a ROSE application, performs roughly the following steps:

- Starting with the AST, it identifies the top-level event methods in the simulation. Different simulator platforms identify event methods in different ways, so this is a simulator-dependent operation. Backstroke must be customized to this extent for each simulator it supports.
- Backstroke applies ROSE's analytical tools to the AST. It uses call graph analysis to identify all functions that are potentially called, directly or indirectly, by the top-level event methods, and interprocedural dataflow analysis to find all variables that may potentially be written. It may also perform static single assignment analysis (Cytron et al. 1991), or aliasing analysis, etc.
- For all of the event methods `E`, Backstroke generates the three derived methods (`E_forward`, `E_reverse`, and `E_commit`), using code inversion algorithms in Backstroke's repertoire.

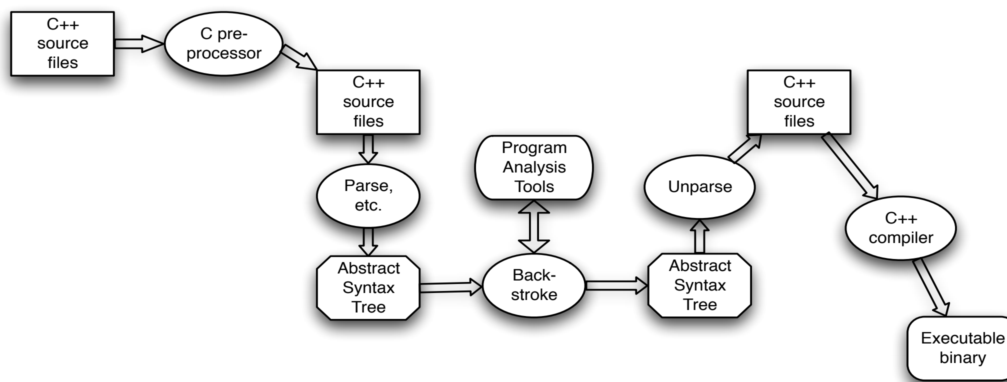

Figure 1: Data flow through the ROSE source-to-source transformation infrastructure and the embedded Backstroke reverse code generator that is embedded in it.

5 TAXONOMY AND TERMINOLOGY FOR INVERSION METHODS

The term "reverse computation" has been used to mean, roughly, any method of event rollback that is based on the source code of the program and attempts to minimize the time and space devoted to "state saving" (including control and state data). The term "perfectly reversible computation" has been used when the amount of state saving required is reduced to zero (Peters and Carothers 2003; Bauer and Page 2007). The trend in recent literature has been to treat "reverse computation" as being in sharp contrast to the older "state saving" methods. We take the view, however, that the contrast is not sharp at all. There is a wide range of methods for accomplishing rollback and we use the term *inversion methods* to cover them all.

We have developed a taxonomy for describing the various inversion methods. The taxonomy may not be complete since it is likely that other methods will be invented in the future. Also, the categories listed here are may overlap. But we believe this classification is a reasonable start.

5.1 Incremental vs. Region-based inversion methods

The first key distinction we make is between *incremental* and *region-based* inversion methods. For present purposes, a *region* of code is roughly defined as any portion of executable source code that has a single entry point. Examples include statements, function bodies, scopes with no labeled statements, and even a single comma expression. Before execution of the region the (relevant) state of the program has some value S_0. During its execution the state is modified a number of times through side effects, creating a sequence of states S_0, S_1, S_2, ... , S_n, where S_n is the final state after execution of the region. The goal of

an inversion method is to restore S_0 given S_n and any additional data saved during forward execution.

i) ***Incremental inversion methods*** restore the initial state S_0 by successively restoring every intermediate program state, starting at S_n, then S_{n-1}, etc. until S_0 is reached. These inversion methods only have to reverse a single side effect at a time, and are hence often easier to implement and can handle a large subset of the C++ language. On the other hand, restoring all intermediate states is not required for implementing event rollback, and there may be optimization opportunities missed by methods requiring the restoration of intermediate states.

There are several examples of incremental inversion methods in the literature. RCC (Perumalla 1999; Carothers et al.1999) implements statement-by-statement reversal of C code for optimistic simulation, while Biswas and Mall (1999) use essentially the same technique on the C source level for debugging purposes. They instrument the forward event method and generate the reverse method using algorithms based on recursion over the syntax of the code. The third column of Figure 2 illustrates the application of these approaches. The RCC approach is elegant, but cannot deal with arbitrary escape constructs (`return`, `break`, `throw`, etc.) RCC relies on the recursive syntactic structure of the code, processing code statement-by-statement, as opposed to using dataflow or interprocedural analyses in its inversion. While syntax-directed inversion is easier to implement, it has the drawback that it does not take advantage of data relationships between separate statements.

A more sophisticated incremental inversion approach, due to Akgul and Mooney (2004), applies incremental inversion at the instruction level as an aid to debugging. In their approach, if a value `x` is destroyed in forward execution, the reaching definition of `x` can be reexecuted in the reverse code to restore it. Alternatively, the value `x` may be recoverable from other variables through uses of `x`, such as "`t = x + 3`". In those cases, `x` is calculated from the other variables (e.g. `t`) that are currently available. These restoration rules are applied recursively, allowing complex reconstruction of overwritten values. Akgul and Mooney's inversion method can invert the code in Figure 2 without having to saving any state data at all, as is shown in the last column.

ii) ***Region-based inversion methods*** restore the initial state S_0 without necessarily restoring any intermediate program states S_1 ... S_{n-1}. These methods can take advantage of the fact that they do not need to reconstruct every intermediate state to generate more efficient reverse code. However, reversing a sequence of side effects is more difficult than reversing a single side effect. Region-based methods may be difficult to apply when the region contains code constructs that are not amenable to static analysis, such as pointer arithmetic, arbitrary memory accesses through `void*` pointers, function calls through dynamic dispatch, and the use of `union` structures.

A *snapshot method* is a region-based method that saves all the state data it is going to save before the forward code for that region begins to execute. Generally it saves a copy of the simulation state variables that are *potentially* modified by the code region, and restores them in the reverse code from the saved copies. Since the saving is done before the code segment starts executing, rather than during execution, performance of such a method depends considerably on the quality of static analysis to determine as small a set as possible of variables or memory locations that *might* be changed. A *binary snapshot* saves a binary image of the static data area and heap. This was done in the original TWOS operating system (Jefferson et al. 1987) and in the Jade system (Unger et al. 1986), but it cannot be applied at the programming language level and is not implementable in Backstroke.

Snapshot methods have several key advantages. If a region of code modifies all or most of its state variables in an information lossy way, then snapshot methods will likely be nearly optimal in space and time. Generally incremental methods can be applied to a larger subset of the C++ language than snapshot methods can, but there are exceptions. Snapshot methods still apply even when the forward execution of the event method produces an uncaught exception, a notoriously difficult problem for other inversion methods. And they can even efficiently invert code that uses asynchronous threading, something almost no other inversion method can accomplish.

iii) **Hybrid methods** combine incremental inversion methods with region-oriented methods to gain the performance benefits of the latter while handling the larger subset of the language afforded by the former. The code to be reversed is divided into maximal regions; a region-based method is used for each region, while incremental inversion is used where region-based methods do not apply. Conceptually, the initial state S_0 is recovered by starting at S_n and making a number of jumps: S_n, S_{n-a}, S_{n-b}, … , S_0.

Original event method E()	Snapshot Inversion	RCC-style syntax-directed incremental inversion	Path-oriented incremental inversion

int a, b;
 	
void E()
{
 if (a > b)
 {
  int t = a;
  b++;
  a = b;
  b = t;
 }
}
	void E_forward()
	{
	 push<int>(a);
	 push<int>(b);
	
	 if (a > b)
	 {
	  int t = a;
	  b++;
	  a = b;
	  b = t;
	 }
	}
		void E_forward()
		{
		 if (a > b)
		 {
		  int t = a;
		  b++;
		  (push<int>(a), a = b);
		  (push<int>(b), b = t);
		  push<bool>(true);
		 }
		 else
		 {
		  push<bool>(false);
		 }
		}
			void E_forward()
			{
			 if (a > b)
			 {
			  int t = a;
			  b++;
			  a = b;
			  b = t;
			 }
			}

	void E_reverse()
	{
	 b = pop<int>();
	 a = pop<int>();
	}
		void E_reverse()
		{
		 if (pop<bool>())
		 {
		  b = pop<int>();
		  a = pop<int>();
		  --b;
		 }
		}
			void E_reverse()
			{
			 if (b > (a - 1))
			 {
			  int t = b;
			  b = a;
			  a = t;
			  --b;
			 }
			}

Figure 2: Example of applying different inversion methods to the same event. Variables `a` and `b` are part of the simulation state. Instrumentation added by Backstroke is in red. Path-oriented methods in column 4 (described in Section 7) can invert this code without saving any additional state.

5.2 Restorative vs. Regenerative inversion methods

Here we borrow the terminology first proposed by Akgul and Mooney (2004). A restorative inversion method copies overwritten values, and later simply restores them, without taking advantage of any mathematical relationships among them. Snapshot methods, as defined earlier, are restorative. Similarly, if restorative inversion is applied incrementally, we get a method similar to the classic incremental state saving (ISS) commonly used in optimistic simulation. In contrast, regenerative inversion methods use relationships among state variables to regenerate parts of the lost state from other parts of the state that are still available or that contain the same information.

RCC and Akgul and Mooney's incremental inversion methods are both regenerative inversion methods. However, RCC only takes advantage of the relationships readily available in a single assignment. For example, an assignment such as `x-=b` can be reversed by RCC as `x+=b`, but a simple swap operation performed as three assignments cannot be handled. On the other hand, Akgul and Mooney's approach analyzes dataflow relationships between statements and can generate much more efficient reverse code, as illustrated by the last column in Figure 2. Even though these two methods are both regenerative incremental methods, they behave very differently due to differences in the power of the program analysis they use. Using more powerful (and expensive) program analyses allows for more efficient regenerative inversion methods. For example, a function that inverts a matrix over a finite field is

technically invertible without state saving, but in order to perform such inversion automatically the program analysis used would have to be powerful enough to compute eigenvalues.

5.3 Combining different inversion methods

Backstroke is designed to allow different inversion methods to be used in different parts of the code. The outer scope of an event method may use an incremental inversion technique, while a snapshot method might be used for an inner scope of the same event body. Alternatively, different methods could be used on different variables, different function bodies, or different types, e.g. a snapshot method for the scalar and class variables, but an incremental method on an element by element basis for array variables. The capability to mix and match inversion methods calls for an automated decision mechanism to pick the most efficient inversion method for each code segment. Currently Backstroke cannot do this, but we intend to assemble a library of extensible performance models that can be used this way.

6 C/C++ LANGUAGE ISSUES

6.1 Inverting function calls

Inverting function calls is imperative for any general inversion framework. Consider a simple illustrative example of a caller and a callee:

```
int f(int x, MyClass* obj) { ... }
void E()
{ ...
  result = f(n, p);
  ...
}
```

The function `f()` is called inside the event method `E()` that is to be inverted. `f()` has a pointer parameter, a value parameter, and a return type, and it also may modify the simulation state arbitrarily. In `E_reverse()` we must correctly restore all simulation state modified by `f()`.

A simple approach to inverting the call to `f()` is simply to inline it. That yields efficient reverse code because calling context and data flow information may be used to specialize `f()`'s inversion. Unfortunately this is almost always impractical because if inlining is recursively applied to all functions there may simply be too much code. And, of course, recursive functions cannot be inlined.

A more general approach to inverting the call to `f()` is to treat it somewhat like an event method, i.e. to generate `f_forward()`, `f_reverse()`, and `f_commit()` and call them inside `E_forward()`, `E_reverse()`, and `E_commit()` respectively. In addition, `f_reverse()` must undo all changes to the simulation state done by `f_forward()`, *regardless of the calling context*. Note that since C++ does not specify the order of argument evaluation in calls to `f()`, Backstroke chooses a canonical order so that side effects in argument evaluation can always be undone in proper reverse order if necessary.

Note that `f_reverse()` has certain preconditions: the simulation state when `f_reverse()` is invoked must be equivalent (as defined in section 7.5 below) to the simulation state immediately after `f_forward()` was executed. This is simple with incremental methods, since they restore all intermediate simulation states anyway. But if a region-based method is to reverse a function call using its reverse function, the code before and after the function call must be treated as separate regions because of the additional intermediate state that must be restored. This applies to implicit function calls such as constructors, destructors and assignment operators if they are not inlined. It is also possible for region-based inversion methods to handle a function call without calling a reverse function by applying inter-procedural analysis. For example, the snapshot method implemented in Backstroke uses interprocedural dataflow information to find the variables modified in the entire call tree of the event method.

6.2 Signatures of reverse functions

When a function `f()` is "factored" into `f_forward()`, `f_reverse()`, and `f_commit()`, the return type and arguments of `f_forward()` are the same as those of `f()`. `f_commit()` takes no arguments and returns `void` since it simply deallocates the object on the rollback stack and commits delayed I/O. The signature of `f_reverse()`, however, requires more consideration. What should the return type of `f_reverse()` be? What parameters should it take? `f_reverse()` need not take `f()`'s return value as a parameter because at the point at which `f_reverse()` is invoked, the calling context has reversed the simulation state to the point immediately after the execution of `f_forward()`. The original return value of `f()` has already been incorporated into the simulation state and may not be readily available to be passed as an argument to `f_reverse()`. Furthermore, even if the return value of `f_forward()` were available from the calling context in some cases, adding it to the function signature would mandate that the return value be restored within *all* calling contexts, even where it is not readily available.

In choosing the formal parameters of `f_reverse()`, we recall that the goal is just to undo the side effects of `f_forward()`. Consider a value parameter, such as "`int x`" in our example. Even if such a parameter is modified inside `f()`, these changes are not reflected within the scope of the caller. One could make the argument that `x` should be a formal parameter to `f_reverse()`, because the value of `x` may be used to improve the efficiency of the inversion. But the value of the actual parameter `x` may not be readily available within the calling context outside of `f_reverse()`. Hence adding "`int x`" to the signature of `f_reverse()` adds an additional variable that must be restored in every context that calls `f()`. The simplest and most efficient choice is to omit *value types* from the formal parameters of `f_reverse()`, i.e. types recursively composed of only scalars, without any pointers or reference types.

Types that contain pointers or references must be kept as arguments to `f_reverse()`. In our example, data may be modified by `f()` through the parameter "`MyClass* obj`", and `f_reverse()` must be able to undo this modification.

6.3 Virtual function calls

When virtual member functions are invoked through a pointer or reference, C++ dispatches the call based on the object's runtime type. In the example below if variable `aa` is declared to be of type `A`, then a call to `aa->f()` may modify `B::x` or `C::y`, depending on the runtime type of `aa`.

```
class A {
   public:
   virtual void f();
};
```

```
class B : public A {
   double x;
   public:
   void f(){ x=7; }
};
```

```
class C : public A {
   double y;
   public:
   void f(){ y=3; }
};
```

Correct inversion of code involving calls to `aa->f()` requires Backstroke to mirror C++'s native dynamic dispatch. This is currently implemented in two ways in Backstroke. For snapshot methods, each class object that must be saved is downcast to its most concrete subtype beforehand. Testing for the dynamic type of a pointer is done either using `dynamic_cast` or using the `typeid` keyword. Due to the runtime overhead of these type detection methods, one can automatically insert an identifier variable as a member in each relevant class, allowing a check of the dynamic type with a simple member comparison.

In incremental methods, the function `f()` would be inverted by generating `f_forward()`, `f_reverse()`, and `f_commit()`. We must do this for every implementation of `f()` that appears in the class hierarchy. Those generated functions are themselves declared `virtual` so we can take advantage of the compiler's own dynamic dispatch mechanism to call the correct generated function for the runtime type of the corresponding object.

6.4 The `delete` keyword and memory deallocation

Consider an event method that contains the code "`delete tPtr`" that deletes an object of type `T`. For a standards-conforming C++ program this statement is irreversible and must be delayed until commit time. However, we cannot simply move the statement "`delete tPtr`" to the commit method. In C++, the `delete` keyword performs two distinct functions – destruction and deallocation. Destruction is a call to `T`'s destructor method and recursively to the destructors for `T`'s members and superclasses in canonical order. These destructors may modify the simulation state (e.g. through static variables) and hence the correctness of the event method would be compromised if their execution were delayed until commit time. Therefore destruction must be done in the forward method while only deallocation is moved to the commit method. Fortunately, C++ provides a means to do this: the destructor of `T` may be called explicitly in the forward event, and `operator delete` may be called in the commit method to deallocate `T`'s storage without invoking destructors, as shown in the following code example:

```
T* tPtr;

void E() {             void E_forward() {                  void E_commit() {
   delete tPtr;           tPtr->T_destruct_forward();         operator delete(tPtr);
}                      }                                   };
```

The `E_reverse()` method must precisely undo the side effects of the `E_forward()` method. Accordingly the reverse method must call the *reverse destructor*. A destructor is treated like a normal function call, so the reverse destructor must mirror the recursive invocation order of destructors mandated by C++. The reverse destructors of member fields and superclasses must be all called, and in the proper reverse order.

6.5 The `new` keyword and memory allocation

Analogously to `delete`, the `new` keyword performs two distinct actions in C++: allocation and construction. Since neither of these is irreversible, the forward method may include invocations of `new`. If the forward method contains "`tPtr = new T()`", then in the reverse method we must first reverse the construction of `T` and then `deallocate T`. Simply calling "`delete tPtr`" in the reverse method is incorrect because C++ does not guarantee or require that `T`'s destructor be the inverse of `T`'s constructor (and frequently in real software it is not). Instead, we must treat each constructor like a function call and in the reverse function we must call *reverse constructors*, rather than destructors. As mentioned above, C++ allows deallocation without destruction through the keyword `operator delete()`.

```
void E() {               void E_forward() {         void E_reverse() {
   tPtr = new T();          tPtr = new T();           tPtr->T_constructor_reverse();
}                        }                            operator delete(tPtr);
                                                    }
```

Note that the handling of `new` and `delete`, discussed above, correctly works even with custom allocators. It is a common practice for high-performance software such as simulations to use custom memory pool allocators in order to mitigate the cost of a kernel call during dynamic memory allocation.

6.6 Pointer arithmetic and `void*` pointers

The arbitrary use of pointers in C++ code causes a number of challenges with inversion because the locations being modified may not be statically resolvable. For example, pointers of type `void*` can point to any object, and their true type is not known statically. Similarly, pointer arithmetic allows the access and modification of arbitrary runtime-computed memory locations. Handling these constructs is important because they commonly appear in real code. For example, STL iterators for the `std::vector<T>` container type are usually implemented through pointer arithmetic.

Incremental inversion methods can reverse arbitrary pointer operations because they do not rely on statically determining the state that was modified. Consider the code "`ptr += n; *ptr = 3`" where `ptr` is a pointer of type `int*` and `n` is an integer. This code could modify an arbitrary location in memory because the value of `n` is not known statically. However, in incremental inversion, the assignment "`*ptr = 3`" can be reversed like any other assignment, since at that point in the reverse code `ptr` is guaranteed to have been restored to its corresponding value from the forward code, whatever it may have been. Similarly, `void*` pointers must always be cast to a concrete type before they are used; hence in the reverse code we can always reverse accesses through concrete pointer types and never have to worry about handling `void*` pointers.

However, the performance of region-based methods in the presence of pointer arithmetic and `void*` pointers depends on the strength of the static analysis used. Aliasing analysis or symbolic execution may be able to determine that certain locations are not modified even in the presence of arbitrary pointer constructs, whereas a simple snapshot method may be forced to save the entire contents of memory space even in the presence of simple pointer arithmetic if the pointer writes cannot be analyzed. In the absence of pointer arithmetic, a pointer value can only be obtained through dereferencing a variable, copying another pointer, or memory allocation; chains of such operations are highly amenable to static analysis. We hypothesize that region-based methods will perform best on pointer code without pointer arithmetic or `void*` pointers, whereas incremental methods will be necessary to reverse arbitrary pointer code.

6.7 Saving and restoring simulation state data in C++

Only side effects on simulation state data are subject to rollback, and the place to start to identify that is with the event methods. Backstroke must contain a filter module for each simulator it supports to provide a way to identify the event methods. The state variables then are, to a first approximation, all the variables and dynamically allocated data structures that are reachable by any event method, or any function called by one directly or indirectly. Stack variables are not included because they do not persist between events, and cannot represent any feature of the physical system being simulated.

But this is not precisely correct, because an event method may request a service from the underlying simulator, which may cause the simulator to modify its own internal state (perhaps something as simple as a counter of the calls to that service). The simulator's own variables are *not* part of the simulation state, and are not subject to saving and restoration during rollback. For this reason Backstroke needs a second filter to distinguish variables or data structures that are not part of the simulation state even though they are modified during the execution of an event. As of this writing Backstroke requires the user to specify a filter written in C++ that distinguishes simulator state variables from model variables. In the future, we plan to make this process more user-friendly by providing `#pragma` directives to mark variables, methods, functions, and user-defined types that are not part of the model's state or evolution.

We also have to define what it means to save and restore a state. Consider S_1 to be the state of an LP just before the execution of event `E`. If `E` is later rolled back to a state S_2 it is not necessary that S_2 be bit-for-bit identical with S_1. However it is required that S_2 be *equivalent* to S_1 in the sense that *the future output of the simulation must be identical regardless of whether the LP is in state S_1 or S_2*. The "output of the simulation" means the output that represents the behavior of the physical system being simulated and not, for example, incidental output that simply documents the run, or reflects runtime performance data, or is otherwise unrelated to the system being modeled. Actual binary pointer values also do not have to be identical between S_1 and S_2. The two states can in principle differ on all pointer values as long as the pointer graphs are isomorphic and the contents of the nodes to which they point are equivalent.

Saving and restoring simulation state data seems at first glance to be a very straightforward operation, and it is for scalar types in C++. The basic pattern for saving and restoring a variable `s` of scalar type `ST` is illustrated in the following code:

```
// Save s                                    // Restore s
ST* scopy = new ST(s);                       s = *scopy;
                                             delete scopy;
```

But it becomes a more complex issue for arrays, structs, class types, and pointer type data. In this section we will describe some of the issues, with emphasis on their intersection with semantic properties of C++.

a) ***Arrays:*** There are two basic strategies for saving and restoring array data. With *atomic copying* the entire array, including all element values, is copied as a unit. With *compound* or *element-wise copying* the array is treated as a compound object, and its elements are saved and restored separately. With snapshot methods atomic copying is natural, but with other methods the choice between atomic or compound copying is very important. If the event is likely to modify most of the elements anyway, then atomic copying is likely to be faster and take less space. But if only a small fraction of the elements will be changed in the event, then it makes sense to use compound copying, since most of the array does not have to be saved and restored at all. Of course with either atomic or compound copying if the elements are of non-scalar types we must (recursively) consider how to save and restore them.

The basic pattern for doing an atomic copy of an array `a[]` of a scalar type `ST` is illustrated in the following code pattern:

```
// Save array a                              // Restore array a
const int n = sizeof(a)/sizeof(a[0]);        for (int ii=0; ii<n; ++ii)
ST* acopy = new ST[n];                         a[ii] = acopy[ii];
for (int ii=0; ii<n; ++ii)                   delete[] acopy;
  acopy[ii] = a[ii];
```

This must be elaborated somewhat, however, if the array elements are not scalars, as we discuss below.

b) ***Class and struct types:*** Copying class-type (including struct-type) state variables in Backstroke opens some non-obvious semantic and design questions. For a variable `c` of class type `CT` the first thought would be to mimic the simple save and restore code used above for scalar types, e.g.:

```
// save c                                    // restore c
CT* ccopy = new CT(c); // (1)                c = *ccopy;        // (2)
                                             delete ccopy;      // (3)
```

But in C++ line (1) invokes the *copy constructor* for type `CT`, line (2) invokes the *assignment operator* for `CT`, and line (3) invokes the *destructor* for `CT`. Those Big Three functions may be implemented by the programmer of the `CT` class, or default implementations may be used. But either way it is essential that together they combine to make a valid save and restore of the state contained in variable `c`. This is relatively straightforward when the data fields of class `CT` are scalars, structs, class types or arrays, but becomes complex when they are pointer types. In that case the copy performed by the copy constructor in line (1) and the assignment in line (2) must be full aliasing- and cycle-aware deep copies. The destructor invoked in line (3) must be a full aliasing- and cycle-aware deletion algorithm as well. Together the code fragment represented by lines (1) through (3), which constitute a save followed by a restore, must be equivalent to no-op in the sense that the state of the program before the save is *equivalent* to the state after the restore.

When an event is not rolled back, the execution of the forward method is eventually followed by the commit method. When the forward method copies a class type variable, the commit method must delete the copy, and that leads to the following pattern being executed.

```
// save c                                    // commit c
CT* ccopy = new CT(c);                       delete ccopy;
```

These lines are the same as lines (1) and (3) above, but without the intervening invocation of the assignment operator. These two lines in sequence must also be equivalent to a no-op.

Of course it is not generally possible to ascertain statically whether or not the Big Three functions for a class `CT` have these required properties, and in many cases they may not. Programmers frequently implement those functions with less than full and deep copies if the intended use does not require it. This issue again highlights the robustness of incremental methods; since incremental methods only reverse a single side-effect at a time, they need only store scalar variables and do not rely on correct implementation of the Big Three functions.

As of this writing, region-based methods in Backstroke simply use the Big Three functions as given in the class definition, trusting that they have the required properties. But we intend to implement a more sophisticated strategy in the future. Backstroke will leave the Big Three functions in place and define alternate interfaces for copy, assignment and delete methods. If the programmer implements those interfaces, Backstroke will trust that they have the required properties. If not, Backstroke will attempt to generate valid implementations, and if successful will use them. But if Backstroke cannot generate provably valid implementations (for example, when a correct copying routine cannot be automatically generated in the presence of a pointer-type member variable that is modified through pointer arithmetic), it will report an error.

c) ***Aliasing:*** Aliasing occurs when a single memory location is reachable through two or more state variables, for example when two pointers point to the same value. This poses a challenge to inversion in two ways: it weakens the accuracy of the analyses used by inversion methods and it potentially causes redundant data to be saved during execution of the forward method. Snapshot methods, for instance, may save the same structure twice if it is referenced by two different pointers. Alternatively, a snapshot method can insert runtime tests that check for aliasing (at additional cost) and then only save each structure once.

7 STATUS AND FUTURE WORK

The Backstroke framework allows composing different inversion methods by partitioning the abstract syntax tree (AST) of the input function. A particular inversion method can be applied to the entire function body, to a particular scope, or even to an individual statement. Each inversion method is given a subtree of the AST, which it may invert as it sees fit, including the invocation of other inversion methods. For example, an incremental inversion method may choose to invoke a snapshot method when it encounters a while-loop.

The Backstroke library presently contains two incremental methods and two region-based methods. Our first incremental method is an implementation of Akgul and Mooney's regenerative algorithm that uses reaching definition and variable use information to recompute destroyed values. If the regenerative search portion of the algorithm is turned off, we obtain a purely restorative incremental inversion method, which simply saves any modified value without an attempt to recompute destroyed values. Backstroke also contains a snapshot method implementation; our snapshot method uses interprocedural analysis to determine all variables potentially modified from the top-level event method being reversed. The analysis correctly (albeit conservatively) handles virtual function calls, function calls through function pointers, aliasing of formal and actual parameters, and recursion.

Backstroke also contains a regenerative region-based inversion method, inspired by Akgul and Mooney's incremental regenerative inversion method. Our region-based method uses a custom analysis built on value graph analysis (Alpern, Wegman, and Zadeck 1988), which in turn is built using static single assignment analysis (Cytron et al. 1991). The approach considers the different path sets in the control flow graph and generates reverse code for each potential path taken. Straight line code is a single path; code with n conditionals (but no loops) may have as many as 2^n paths distinct. Forward and reverse code is generated separately for each path set, and then the separate reverse path sets are reassembled and reintegrated so that the correct reverse code is dynamically executed. The forward code is instrumented to efficiently record the actual path taken. Figure 3 shows the reverse code generated by this region-based method for a small example.

Original event method E()

```
int a, b, c;

void E()
{
  if (a > 0) {
   b = a + 10;
   a = 3;
  }
  if (c == 0)
   c = 5;
  else
   c = 7;
}
```

E_forward()

```
void E_forward()
 {
  int pathNum = 0;
  if (a > 0) {
    push<int>(b);
    b = a + 10;
    a = 3;
  }
  else
    pathNum += 2;
  if (c == 0)
    c = 5;
  else {
    pathNum += 1;
    push<int>(c);
    c = 7;
  }
 }
```

E_reverse()

```
void E_reverse()
 {
  int pathNum = pop<int>();
  if (pathNum & 1)
    c = pop<int>();
  else
    c = 0;
  if (pathNum & 2 == 0) {
    a = b - 10;
    b = pop<int>();
  }
 }
```

Figure 3: Example application of Backstroke's path-oriented regenerative inversion algorithm. The variables `a`, `b`, and `c` are state variables. The forward code is instrumented in red to keep track of the dynamic path taken, and the reverse code uses path information to restore variable values.

To summarize, Backstroke is a framework for the automatic generation of code with the ambitious goal of being able to generate correct and efficient reverse code for essentially any event method written in C++. It is designed to handle a library of many different inversion algorithms and to allow different ones to be applied in different parts or scopes of the simulation code. Because C++ is a very large and object oriented language, new issues arise that have not been dealt with in previous research, especially regarding the treatment of class types, pointers, and polymorphism. Because Backstroke is embedded in the ROSE source-to-source compiler infrastructure we are able to apply advanced program analysis tools developed for other purposes to the task of generating reverse code. Besides expanding our library of inversion methods in the future, we plan to study their space and time performance. A major open problem is how to decide statically which inversion method to use in each event method or scope.

Backstroke is an open source project. We hope to we can attract collaborators from anywhere to help reduce the software engineering and performance barriers to widespread use of optimistic parallel discrete event simulation.

REFERENCES

Agrawal, H., R.A. DeMillo, E.H. Spafford. 1991. "An execution backtracking approach to program debugging," *IEEE Software*, vol. 8, 1991, pp. 21-26.

Akgul, T. and V.J. Mooney III. 2004. "Assembly instruction level reverse execution for debugging," *ACM Transactions on Software Engineering and Methodology*, vol. 13, Apr. 2004, pp. 149-198.

Alpern, B., M.N. Wegman, and F.K. Zadeck. 1988. "Detecting Equality of Variables in Programs," *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, ACM, 1988, pp. 1-11.

Archer, J., J.E., R. Conway, and F.B. Schneider. 1984. "User recovery and reversal in interactive systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 6, Jan. 1984, pp. 1-19.

Bagrodia, R., R. Meyer, B. Park, H. Song, Y. Chen, X. Zeng, J. Martin, M. Takai. 1998. "PARSEC: A parallel simulation environment for complex systems," *IEEE Computer Magazine*, 1998

Bauer, D. W., E.H. Page. 2007. "An Approach for Incorporating Rollback through Perfectly Reversible Computation in a Stream Simulator," *21st International Workshop on Principles of Advanced and Distributed Simulation (PADS '07)*, IEEE Computer Society, 2007, pp. 171-178.

Bishop, P.G. 1997. "Using reversible computing to achieve fail-safety," *Proceedings of the Eighth Int'l Symp. on Software Reliability Engineering*, IEEE Computer Society, 1997, pp. 182-191.

Biswas, Bitan, and R Mall. 1999. "Reverse execution of programs." *SIGPLAN Notices*, vol. 34, 1999, pp. 61-69.

Briggs, J.S. 1987. "Generating reversible programs," *Software: Practice and Experience*, vol. 17, Jul. 1987, pp. 439-453.

Carothers, C. D., K.S. Perumalla, and R.M. Fujimoto. 1999. "Efficient optimistic parallel simulations using reverse computation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 9, 1999, p. 224–253.

Carothers, C. D. 2002. "ROSS: A high-performance, low-memory, modular Time Warp system," *Journal of Parallel and Distributed Computing*, vol. 62, Nov. 2002, pp. 1648-1669.

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

Das, S., R.M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994. "GTW: A Time Warp system for shared memory multiprocessors," *Proceedings of the 26th conference on Winter simulation, Society for Computer Simulation International*, 1994, p. 1332–1339.

Feldman S. I. and C.B. Brown. 1998. "Igor: A system for program debugging via reversible execution," *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging*, ACM, 1988, p. 123.

Fujimoto, R. M.. 2000. *Parallel and Distributed Simulation Systems*, Wiley, 2000.

Jefferson, D. R. 1985. "Virtual Time", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 7, 3, pp. 404-425, July 1985

Jefferson, D. R., Motro, A. 1986. "The Time Warp Mechanism for Database Concurrency Control", *Proceedings of the IEEE 2nd International Conference on Data Engineering*, Los Angeles, CA, Feb-4-6 1986

Jefferson, D. R., B. Beckman, F. Wieland, L. Blume, M. DiLoreto. 1987. "Distributed Simulation and the Time warp operating system," *Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, ACM, 1987, p. 77–93.

Naborskyy, A., R.M. Fujimoto. 2007. "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)*, Jun. 2007, pp. 179-188

Nicol, D. M. and P. Heidelberger. 1995. "On extending parallelism to serial simulators," *Proceedings of the ninth workshop on Parallel and Distributed Simulation*, 1995, pp. 60-67.

Perumalla, K.S., R.M. Fujimoto. 1999. "Source-code transformations for efficient reversibility", CC Technical Report, GIT-CC-99-21, Georgia Institute of Technology, Atlanta, Georgia, USA: 1999.

Perumalla, K. S. 1999. "Techniques for efficient parallel simulation and their application to large-scale Telecommunication Network Models," dissertation, Georgia Institute of Technology, 1999.

Perumalla, K.S. 2005. "uSik A Micro-Kernel for Parallel/Distributed Simulation Systems," *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, IEEE Computer Society, 2005, p. 59–68.

Peters, M., C.D. Carothers. 2003. "An algorithm for fully-reversible optimistic parallel simulation," *Proceedings of the 2003 Winter Simulation Conference (WSC)*, IEEE, 2003.

Quinlan, D. 2011. ROSE, http://www.rosecompiler.org. A listing of ROSE-related papers can be found at http://www.rosecompiler.org/ROSE_HTML_Reference/ProjectPublications.html

Riley, G. F., M.H. Ammar, R.M. Fujimoto, A. Park, K.S. Perumalla, and D. Xu. 2004. "A federated approach to distributed network simulation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 14, Apr. 2004, p. 116–148.

ROSE. 2011. http://www.rosecompiler.org

Santoro, A. and F. Quaglia, 2006. "Transparent Optimistic Synchronization in HLA via a Time-Management Converter," *20th Workshop on Principles of Advanced and Distributed Simulation (PADS '06)*, 2006, pp. 193-200.

SPEEDES. 2011. http://www.speedes.com

Steinman, J. 1992 "SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation," *International Journal in Computer Simulation*, Vol. 2, pages 251-286

Subramania, S., M. Thazhuthaveetil. 2001 "TWLinuX: Operating System Support for Optimistic Parallel Discrete Event Simulation," *High Performance Computing-HiPC 2001*, S. Monien, Burkhard and Prasanna, Viktor and Vajapeyam, ed., Springer, 2001, p. 262–271.

Tang, Y., K.S. Perumalla, R.M. Fujimoto 2006 "Optimistic simulations of physical systems using reverse computation," *Simulation*, v. 82, pp 61-73, 2006.

Unger, B., J. Cleary, G. Lomow, Xining, L., Zhonge. X., Slind, K. 1986. Jade Virtual Time Implementation Manual, 1986.

Wang, X. 2005. "Optimistic Synchronization in HLA-Based Distributed Simulation," *Simulation*, vol. 81, Apr. 2005, pp. 279-291.

Wegman, M. N., Bowen Alpern, F.K. Zadeck. 1988. "Detecting Equality of Variables in Programs," *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, 1988, pp. 1-11

Zelkowitz, M. V. 1973. "Reversible execution," *Communications of the ACM*, vol. 16, Sep. 1973, p. 566.

AUTHOR BIOGRAPHIES

GEORGE VULOV is a Ph.D. student in the School of Computational Science and Engineering at the Georgia Institute of Technology. His research interests include reverse computation, program analysis, and agent-based simulation. His email address is georgevulov@gatech.edu.

CONG HOU is a PhD student of computer science in Georgia Institute of Technology. His research interest includes compilers, optimization, programming languages and reverse computation. His email address is hou_cong@gatech.edu.

DANIEL QUINLAN is the leader of the ROSE project at Lawrence Livermore National Laboratory. His research interests include object-oriented numerical frameworks, semantics-based source code transformations, C++ compiler tools, infrastructure, and design, parallel array classes, parallel data distribution mechanisms, and parallel load balancing algorithms. His email address is dquinlan@llnl.gov.

RICHARD VUDUC is an Assistant Professor in the School of Computational Science and Engineering at the Georgia Institute of Technology. His research lab, the HPC Garage (hpcgarage.org), is interested in parallel algorithms, performance analysis and tuning, and debugging. His email is richie@cc.gatech.edu.

RICHARD FUJIMOTO is Regents' Professor and founding Chair of the School of Computational Science and Engineering at the Georgia Institute of Technology. He has been an active researcher in the parallel and distributed simulation field since 1985. His email address is fujimoto@cc.gatech.edu.

DAVID JEFFERSON is a researcher in scalable simulation at Lawrence Livermore National Laboratory and leader of the Backstroke project. He was co-inventor Time Warp, the first optimistic synchronization method for parallel discrete event simulation. His email address is jefferson6@llnl.gov.