# Source-code Transformations for Efficient Reversibility

## Technical Report GIT-CC-99-21

Kalyan S. Perumalla
kalyan@cc.gatech.edu

Richard M. Fujimoto
fujimoto@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA 30332-0280

# Contents

# Source-code Transformations for Efficient Reversibility

Kalyan S. Perumalla
Richard M. Fujimoto
College of Computing
Georgia Institute of Technology

September 1, 1999

**Abstract**

An approach based on source-to-source compilation is presented for efficiently realizing reversibility of programs, in the context of the **C** language. The implementation of this approach in a portable "reverse **C** compiler" called `rcc` is described. The compiler takes arbitrary **C** functions as input, and renders them reversible, and, in addition, automatically generates their reverse functions. The system is sufficiently general to correctly handle complex data types and to handle complex control flow such as recursion and inter-mixing jump instructions. The compiler is designed to be general-purpose, to be used in any application requiring efficient reverse execution capability. A simple macro interface is used to help retarget the compiler to different application domains. As a case study, the application of `rcc` to optimistic parallel simulation is presented. Optimizations, such as *message-send slicing*, are presented that help in reducing the overheads that may be added for achieving reversibility.

# 1 Introduction

Reversibility is a useful property in many applications, such as debugging, database recovery, and optimistic parallel simulation. In these applications, the need arises to undo erroneous or unwanted changes to the state of the program. Reversibility of a program requires that information is never destroyed during the program execution. Programs written in popular general-purpose languages, such as **C** and Pascal, tend to be irreversible in nature. Due to the very nature of the language constructs, some information is destroyed as the program executes, making the program difficult to execute in reverse. However, any program can be made reversible by carefully augmenting it in such a way that the information being destroyed by the program is actually saved on the side using additional non-intrusive instructions. The augmented program thus not only retains the semantics of the original program, but also becomes reversible. Such an augmenting approach is described here, which can be used to transform any given (irreversible) program into a semantically equivalent reversible program. The automatic generation of the reverse program is also described. Such automated means for reversibility, as presented here, is useful and important since the application code can be too complex for manual transcription and optimization by human inspection. In the presentation that follows, the programs are assumed to be written in the **C** language. However, the techniques are equally applicable to other similar procedural languages (*e.g.* Pascal).

## Relation to Perfectly Reversible Computing

It may be noted that specialized languages can indeed be carefully defined in such a way that their constructs never permit any destruction of information during program execution. In other words, programs written in such languages are perfectly reversible. However, such languages are not the focus of this work, since they are not commonly used. Moreover, there is an important distinction between the work described here and other related work on perfectly reversible computing. In perfect reversibility, a forward function `f()` and its reverse `rf()` are defined to be commutative on the program state $S$:

$$\mathtt{rf}(\mathtt{f}(S)) \equiv S \equiv \mathtt{f}(\mathtt{rf}(S))$$

In other words, `f()` and `rf()` are defined to be perfect inverses of each other[1]. However, in applications of our interest, such as parallel simulation, the property of commutativity is in fact not necessary. Instead, only the left half of the previous identity is sufficient for practical purposes:

$$\mathtt{rf}(\mathtt{f}(S)) \equiv S, \text{ but } S \not\equiv \mathtt{f}(\mathtt{rf}(S))$$

For example, consider the following fragment in which an irreversible program fragment `f'()` is transformed to a reversible version `f()` by adding a bit variable `b`, while `rf()` is the reverse of `f'()`:

| Original | Reversible | Reverse |
|---|---|---|
| `int x = 0, y = 1;`<br><br><br>`f'()`<br>`{`<br>    `if( x > y )`<br>        `y += x;`<br>    `else`<br>        `x += y;`<br>`}` | `int x = 0, y = 1;`<br>`bit b;`<br><br>`f()`<br>`{`<br>    `if( b = (x > y) )`<br>        `y += x;`<br>    `else`<br>        `x += y;`<br>`}` | `int x = 0, y = 1;`<br>`bit b;`<br><br>`rf()`<br>`{`<br>    `if( b )`<br>        `y -= x;`<br>    `else`<br>        `x -= y;`<br>`}` |

In this, the state $S = \{\mathtt{x}, \mathtt{y}, \mathtt{b}\}$. It is clear that $\mathtt{rf}(\mathtt{f}(S)) = S$. This is because `x` and `y` are restored to their original values, and `b` (being an added variable) remains in a *don't care* state. However, $\mathtt{f}(\mathtt{rf}(S)) \neq S$.

Perfect reversibility supports the power of the retracing of execution by starting from the final state and reverse computing towards the initial state. Such power is not always necessary for the practical applications such as optimistic simulation, in which forward reversibility is sufficient. The reduced power of this type of "half" reversibility can translate to savings in runtime and memory overheads.

## Compilation vs. Interpretation

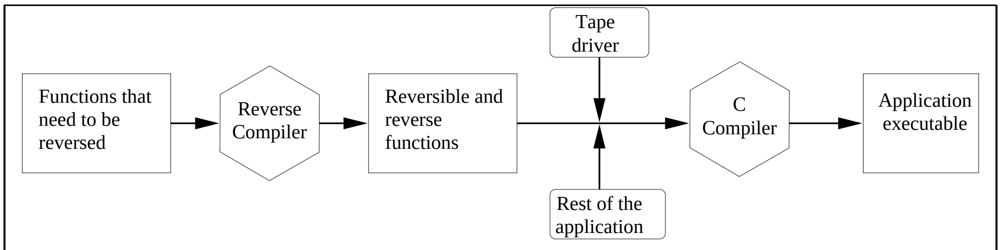
In the case of high-performance applications of reversibility, the undo capability is required to be of low overhead with regard to memory and time. For example, in optimistic parallel simulations that execute hundreds of thousands (or even millions) of events per second, the rollback overhead per event must be kept very low. The low overhead requirements of high-performance applications precludes the use of interpretation-based approaches to reversibility. In such applications, the benefits of compiled reverse code are highly desirable. On the other hand, in other applications such as interactive debugging, in which flexibility and control are important, interpretation-based approach to reverse execution are applicable, as described in [1]. The source-to-source translation system described here uses the compilation approach and avoids interpretation, to keep the runtime overheads low.

## Compilation Architecture

The compilation architecture for reversible applications is illustrated in the following figure. The portion of the application that needs to be made reversible is fed through the reverse **C** compiler,

[1] If $S$ is $n$ bits long, the number $N$ of distinct programs $g() : S \to S$ is given by: $N = \sum_{k=1}^{2^n} \binom{2^n}{k}\binom{2^n}{k-1}$. The functions `f()` and `rf()` are two among them.

which produces reversible versions of its input functions along with their reverse functions. The reversible and reverse functions that are output are combined with the application's customized tape driver (described later in section 2) along with the rest of the application. A conventional **C** compiler is used to compile all of them together to result in the application executable.



Architecture for applications using reverse compilation

The rest of the document is organized as follows. The compilation techniques are described in section 2, followed by section 3 that provides an overview of the implementation of the techniques in the `rcc` compiler. The use of `rcc` in optimistic simulation applications is described as a case study in section 4, followed finally by a summary in section 5 and list of open issues in section 6.

# 2 Compilation

The compilation is performed in three phases, as illustrated in the following figure. Each of these phases is described in detail in this section.

Reverse Compilation Phases

In the rest of the document, the term `s` represents any simple or compound statement, while `rs` denotes its reverse. Also, a bit/byte tape data structure that follows LIFO (last-in-first-out) discipline is assumed to be maintained at runtime for reversibility. A `SAVE(x)` operation results in appending the value of the *lvalue* `x` to the end of the tape, and `RESTORE(x)` retrieves the current value from the end of the tape and stores it into the *lvalue* `x`. The number of bits stored or retrieved is equal to `8*sizeof(x)`. A `SAVE_BITS(x,n)` operation results in saving only the lowest $n$ bits of the integral *lvalue* `x`. Similarly, `RESTORE_BITS(x,n)` results in restoring only $n$ bits into the integral *lvalue* `x`.

## 2.1 Normalization

For convenience, the input program is first translated into another program that is semantically equivalent but uses a reduced set of language constructs. For example, all jump instructions (`case`, `break`, `return`) are translated to equivalent forms of the `goto` statement. Another important example is the treatment of initializer expressions in variable declarations. These initializer expressions can be moved out of the declarations into actual assignment statements in the front of the code in which the variables are defined. The resulting separation of declaration from assignment is quite useful in later phases of compilation.

This phase, called the *normalization* process, is performed as the first step in the compilation process. The advantages of normalization are that it is simpler to make the normalized code reversible, and then, generating the reverse code becomes easier.

### 2.1.1 Declarations

Type declarations as well as global and static variable declarations are retained unchanged, since they are initialized at the beginning of program execution. Local variable declarations are also retained unchanged, except for moving their initializer expressions into explicit assignment statements, as illustrated in the following.

| Original | Normalized |
|---|---|
| `{`<br>`  double x = 10, y = log(x)+1;`<br>`  ...`<br>`}` | `{`<br>`  double x, y;`<br>`  {`<br>`    x = 10;`<br>`    y = log(x)+1;`<br>`  }`<br>`  ...`<br>`}` |

### 2.1.2 Side Effect Expressions

All expressions which have side effect operators are converted to equivalent set of expressions which do not have side effects, with one exception: each void function call appears as a statement of its own, and each non-void function call appears as the only term in the right hand side expression of an assignment statement. The elimination of side effects in expressions is easily performed using standard algorithms, which add new temporary variables to hold intermediate values to make the expressions side effect free. Thus, expressions such as `if` and `while` conditions, function call arguments, array indices, pointer arithmetic, etc., must be converted to be free of side effects. Care must be taken in the case of loop conditions (`while`, `for` and `do-while` conditions). For example, the side effect operations, if any, in a `while` condition must be duplicated: (1) just before entering the loop, and (2) at the end of the loop body.

### 2.1.3 Function Call

All function calls are normalized such that a call to a `void` function appears as a statement by itself, and a non-`void` function call appears as a single righthandside term of an assignment statement:

$$\mathit{lvalue} \texttt{ = } \mathit{function\text{-}call}\texttt{(}\cdots\texttt{)}\texttt{;}$$

### 2.1.4 Arithmetic Expressions

To prevent loss of numerical precision during computation, all arithmetic expressions are normalized such that each value or variable is promoted to a data type of the next higher precision. This, of course, assumes that a special data type is available to the compiler that is of higher precision than the highest precision type used in the input functions. Otherwise, software emulation of higher precision is necessary.

### 2.1.5 `for` Statement

The `for` statement is normalized into its trivial `while` form, as follows. A missing `test` expression is substituted with `1`. Since `s` can be a non-compound statement, it is necessary to house it in a compound statement along with `incr` expression.

| Original | Normalized |
|---|---|
| `for( init; test; incr )`<br>`    s` | `{`<br>`  init;`<br>`  while( test )`<br>`  {`<br>`      s`<br>`      incr;`<br>`  }`<br>`}` |

### 2.1.6 `return` Statement

Multiple `return` statements are converted into a single `return` statement using the following trivial normalization which uses a temporary local variable for holding the returned value.

| Original | Normalized |
|---|---|
| <pre>T foo(...)<br>{<br>  ...<br>  return x;<br>  ...<br>  return y;<br>  ...<br>}</pre> | <pre>T foo(...)<br>{<br>  T t;<br>  ...<br>  {t = x; goto label;}<br>  ...<br>  {t = y; goto label;}<br>  ...<br>label:<br>  return t;<br>}</pre> |

In the case of a `void` function, the variable can be eliminated, as illustrated in the following.

| Original | Normalized |
|---|---|
| <pre>void foo(...)<br>{<br>  ...<br>  return;<br>  ...<br>  return;<br>  ...<br>}</pre> | <pre>void foo(...)<br>{<br>  ...<br>  goto label;<br>  ...<br>  goto label;<br>  ...<br>label:<br>  return;<br>}</pre> |

After the normalization, at most one `return` statement will remain in any function. This ensures a single exit point for each function, implying a single entry point for its reverse. In the transformation and reversal phases, no further treatment is needed for the `return` statement. Hence it is retained unchanged in the transformed code, and eliminated in the reverse code. Note that the `label` name must be generated to be unique in the function scope.

### 2.1.7 `continue` Statement

The following trivial normalization is performed on a `continue` statement by translating it into an equivalent `goto` statement. Note that the `label` name must be generated to be unique in the function scope.

| Original | Normalized |
|---|---|
| <pre>{<br>  ...<br>  continue;<br>  ...<br>}</pre> | <pre>{<br>  ...<br>  goto label;<br>  ...<br>label:<br>}</pre> |

In the case of `while`, `for` and `do-while` statements, the position of `label` must be carefully placed so that it is coordinated with the increment statement that is instrumented into the transformed code.

### 2.1.8 `break` Statement

The following trivial normalization is performed on `break` statement by translating it into equivalent `goto` statement. Note that the `label` name must be generated to be unique in the function scope.

| Original | Normalized |
|---|---|
| `{`<br>`  ...`<br>`  break;`<br>`  ...`<br>`}`<br>`...`<br>`s` | `{`<br>`  ...`<br>`  goto label;`<br>`  ...`<br>`}`<br>`label:`<br>`...`<br>`s` |

Note that this normalization is *not* applied to the `break` statements that belong to a `switch` statement. The `break` statements inside a switch statement are handled differently, as described in sections 2.1.9 and 2.2.7.

### 2.1.9 `switch` Statement

The `switch` statement can be treated in two different ways for the purposes of reversibility. In the first approach, it can be normalized into equivalent set of `if` and `goto` statements, as illustrated in the following.

| Original | Normalized |
|---|---|
| `switch( expr )`<br>`{`<br>`  case c1:`<br>`    s1`<br>`    break;`<br>`  case c2:`<br>`    s2`<br>`    /*fall-thru*/`<br>`  case c3:`<br>`    s3`<br>`    break;`<br>`  default:`<br>`    s4`<br>`    break;`<br>`}` | `{`<br>`  int c = expr;`<br>`  if( c == c1 ) goto label1;`<br>`  if( c == c2 ) goto label2;`<br>`  if( c == c3 ) goto label3;`<br>`  goto labeld;`<br>`label1:`<br>`  s1`<br>`  goto end;`<br>`label2:`<br>`  s2`<br>`  /*fall-thru*/`<br>`label3:`<br>`  s3`<br>`  goto end;`<br>`labeld:`<br>`  s4`<br>`  goto end;`<br>`end:`<br>`}` |

This normalization is a straight-forward application of the semantics of a `switch` statement, which states that every case value acts as a label which becomes a target of a *computed goto* statement. Note that saving and restoring operations for the value of `c` will be introduced in the transformation phase of compound statements, as described in section 2.2.5.

In the alternative approach, a `switch` statement can be transformed into another equivalent, but reversible, `switch` statement, as discussed later.

### 2.1.10 Jumps across Nested Blocks

Compound statements (block scopes) with normal (fall through) entry and exit are treated in the transformation phase, as discussed later in section 2.2.5. However, in the case of unusual entry and exit via jump statements such as `goto`, the values of all the local variables that are about to go out of scope must be saved before the jump takes place. For this, the appropriate `SAVE()` calls are inserted before each `goto`, as illustrated in the following.

For example, a `goto` statement that takes the control out of a block scope must result in saving all the local variables in the block scope before transfering the control. Moreover, since it is possible that a single `goto` statement in a deeply nested block can cut across several block scopes, the local variables of all those blocks must also be saved before the jump is effected.

*This is exactly analogous to the manner in which $C^{++}$ generates calls to destructors on all local variables at all levels of block scope that are about to go out of scope as a result of any single* `goto` *statement[3].*

<table>
<tr><th>Original</th><th>Normalized</th></tr>
<tr><td><pre>
{/*C1*/
  int x;
  ...
  {/*C2*/
    int y;
    ...
    {/*C3*/
      int z;
      ...
      goto label;
      ...
    }
  }
  ...
label:
  ...
}
</pre></td><td><pre>
{/*C1*/
  int x;
  ...
  {/*C2*/
    int y;
    ...
    {/*C3*/
      int z;
      ...
      {SAVE(z); SAVE(y);}
      goto label;
      ...
    }
  }
  ...
label:
  ...
}
</pre></td></tr>
</table>

In the previous illustration, the `goto` statement would result in all the variables of `C2` and `C3` (`y` and `z`) going out of scope. Hence, their values are saved before the jump is executed. Note that the variable `x` is *not* saved since it remains in scope even after the jump, and hence will be saved by instructions generated by the normal transformation phase (see section 2.2.5).

This normalization must be performed *after* the `return`, `continue`, `break` and `switch` statements are normalized. This is necessary in order to take into account the new `goto` statements that may be generated during normalization of those statements.

## 2.2 Transformation

At the end of the normalization, the following are true:

- At most one assignment operator remains in any given expression statement. Moreover, both the lefthandside and righthandside of the assignment operator are free of side effects.
- All `for` statements are eliminated by translating them into equivalent `while` statements.
- The condition expressions of `if`, `switch`, `while` and `do-while` statements are free of side effects.
- The only type of jump statement remaining is the `goto` statement (with the exception of a single `return` statement at the end of the function body).
- There is at most one `return` statement in any function, which will be towards the end of the function body. Moreover, in the case of non-`void` functions, the return expression consists of exactly one variable.
- The relationship between unusual jumps across block scopes and the local variables in those blocks can be ignored in the transformation, since they are taken care of in the normalization. Thus, only normal entry into (and exit out of) block scopes need be considered.

In the transformation phase, the input code is made reversible, and in addition, its reverse is also generated. The rules used in the transformation phase are discussed next. These rules are applied starting with each function definition in the input, and every statement in the function is recursively transformed.

### 2.2.1 Expression Statement

In **C**, constructive operators are closed with respect to inversion, including the following:

| **Constructive Operator** | **Inverse Operator** | **Remarks** |
|---|---|---|
| `++` | `--` | postfix to prefix; mutual inverses |
| `+=` | `-=` | mutual inverses |
| `/=` | `*=` | mutual inverses |
| `^=` | `^=` | self inverse |

All other operators are destructive assignments, which require the saving of the value of the operand before it is destroyed due to the operation. These operators include `=`, `%=`, and so on. Since all such operators are normalized into the form *lvalue op rvalue*, they are transformed to save the *lvalue* before the assignment, as follows:

| Original | Transformed | Reversed |
|---|---|---|
| `lvalue op rvalue;` | `{`<br>`  SAVE(lvalue);`<br>`  lvalue op rvalue;`<br>`}` | `{`<br>`  RESTORE(lvalue);`<br>`}` |

### 2.2.2 Simple Function Call

Since all function calls are normalized such that they are one of the forms $function - call(\cdots)$ or $lvalue = function - call(\cdots)$, they are reversed by replacing the function name with the name of its reverse function.

### 2.2.3 Function Pointer Call

To support function calls that are in fact evaluated on function pointer expressions, a hash table of function pointers can be maintained at runtime by the compiler, which is used to map a function pointer to its reverse function pointer. Thus, if the function call is not via a simple function name, but is in fact via an expression that evaluates to a pointer to a function, then the hash table of function pointers is consulted to call the correct reverse function.

| Original | Transformed | Reversed |
|---|---|---|
| `(func-ptr-expr)(...);` | `(func-ptr-expr)(...);` | `(lookup(func-ptr-expr))(...);` |

### 2.2.4 Jump Statement

For each label that is the target of one or more `goto` statements in the input, the following transformation is applied.

| Original | Transformed | Reversed |
|---|---|---|
| `{`<br>`...`<br>`s1`<br>`goto label;`<br>`...`<br>`s2`<br>`goto label;`<br>`...`<br>`s3`<br>`...`<br>`label:`<br>`s4`<br>`...`<br>`}` | `{`<br>`char c;`<br>`...`<br>`s1`<br>`{c=1; SAVE(c);}`<br>`goto label;`<br>`...`<br>`s2`<br>`{c=2; SAVE(c);}`<br>`goto label;`<br>`...`<br>`s3`<br>`...`<br>`{c=0; SAVE(c);}`<br>`label:`<br>`s4`<br>`...`<br>`}` | `{`<br>`char c;`<br>`...`<br>`rs4`<br>`RESTORE(c);`<br>`switch( c )`<br>`{`<br>`case 0: break;`<br>`case 1: goto rlabel1;`<br>`case 2: goto rlabel2;`<br>`}`<br>`...`<br>`rs3`<br>`...`<br>`rlabel2:`<br>`rs2`<br>`...`<br>`rlabel1:`<br>`rs1`<br>`...`<br>`}` |

This transformation works by using a variable `c` to remember which of the `goto` statements was actually executed to reach the labeled statement in the forward execution. During reverse execution, this information is used to jump back to the point that immediately precedes the previously executed `goto` statement. To perform the reverse jump, the `goto` statements of the forward code are replaced by labels, which are reached using a *computed goto* in the reverse code.

This transformation is capable of working correctly in the presence of multiple interdependent jump statements. It also correctly handles multiple visits to the same target label during the same function call. The `SAVE()` calls inserted before the `goto` statements are necessary to handle multiple visits to the same jump statement.

Note that the variable name `c` and the names of the labels in the reverse code–`rlabel`*i*–must be generated so that they are unique in the given function body scope. An easy way to do this is to use the label of the `goto` statement given in the original input. Also, note that the variables are declared to be of type `char`, assuming the number of `goto` statements for any label fits one byte (this is usually true human-generated models). Otherwise, a larger integral type, such as `short int`, can be used.

Since the variable `c` is saved before the jump instruction, it need *not* be saved again when the variable goes out of scope, as is normally done for a compound statement as discussed next.

### 2.2.5 Compound Statement

Every compound statement (also called the block scope) is transformed according to the following illustration.

| Original | Transformed | Reversed |
|---|---|---|
| `{`<br>`int x, y;`<br>`s1`<br>`s2`<br>`s3`<br>`}` | `{`<br>`int x, y;`<br>`s1`<br>`s2`<br>`s3`<br>`SAVE(x);`<br>`SAVE(y);`<br>`}` | `{`<br>`int x, y;`<br>`RESTORE(y);`<br>`RESTORE(x);`<br>`rs3`<br>`rs2`<br>`rs1`<br>`}` |

By this transformation, the order of executable statements in the block are preserved in the forward code, but reversed in the reverse code. Since local variables lose their values when they go out of scope, they must be saved towards the end of the block in the forward code, and restored towards the beginning of the block in the reverse code. Hence, state saving calls are automatically added at the end of every compound statement body, so that the values of the local variables are saved for restoration during reverse execution. Abnormal exits from the block scope are treated specially, as described previously in the normalization phase (see section 2.1.10).

Formal parameters to functions are logically similar to local variables of the function body. They are state saved by treating them exactly like the local variables of the function body.

### 2.2.6 `if` Statement

Each `if` statement is made reversible by introducing a variable to remember the result of the `if` condition. This value of this variable is saved after the `if` statement is executed during the forward execution. The saved value is restored during reverse execution, and it is used to recollect which of the true or false branches was executed in the forward execution. The corresponding branch is then executed in reverse. The transformation is as follows.

| Original | Transformed | Reversed |
|---|---|---|
| `if( condition )`<br>`    s1`<br>`else`<br>`    s2` | `{`<br>`  char c = !!condition;`<br>`  if( c )`<br>`    s1`<br>`  else`<br>`    s2`<br>`  SAVE(c);`<br>`}` | `{`<br>`  char c;`<br>`  RESTORE(c);`<br>`  if( c )`<br>`    rs1`<br>`  else`<br>`    rs2`<br>`}` |

In practice, the `SAVE(c)` and `RESTORE(c)` are replaced by `SAVE_BITS(c,1)` and `RESTORE_BITS(c,1)`, since the actual value of `c` is only contained in the lowest bit. Note that the save and restore operations on `c` are actually generated as part of the transformation phase on compound statements, as described in section 2.2.5. Note also that it is necessary to coerce the truth value of the condition to 0 or 1, to make it fit in one bit. The coercion is necessary because **C** permits any non-zero value to qualify as a true value, which obviously does not fit in one bit. Double negation (using the `!` operator twice) is one way to perform the coercion, as illustrated in the transformed code.

A trivial optimization can be performed by eliminating the added variable in case the `if` condition is invariant in both true and false branches of the `if` statement. This is because the same expression can be re-evaluated (without loss of information) during reverse execution to recollect which branch was executed, as illustrated in the following.

| Original | Transformed | Reversed |
|---|---|---|
| `if( condition )`<br>`    s1`<br>`else`<br>`    s2` | `{`<br>`  if( condition )`<br>`    s1`<br>`  else`<br>`    s2`<br>`}` | `{`<br>`  if( condition )`<br>`    rs1`<br>`  else`<br>`    rs2`<br>`}` |

Even though only a single bit is reduced by such optimization per `if` statement execution, such optimization can result in significant reduction in overall memory utilization when the statement is repeatedly executed, say, inside a loop.

### 2.2.7 `switch` Statement

One way to treat `switch` statement is to normalize it into `if` and `goto` statements, as described in section 2.1.9. An alternative technique is to transform it into another `switch` statement, which uses two integer variables, `c` and `d`, as illustrated in the following.

| Original | Transformed | Reversed |
|---|---|---|
| `switch( expr )`<br>`{`<br>`  case v1:`<br>`    s1`<br>`    break;`<br>`  case v2:`<br>`    s2`<br>`    /*fall-thru*/`<br>`  case v3:`<br>`    s3`<br>`    break;`<br>`  default:`<br>`    s4`<br>`    break;`<br>`}` | `{`<br>`  char c = 0, d = 0;`<br>`  switch( expr )`<br>`  {`<br>`    case v1: c=1; d++;`<br>`      s1`<br>`      break;`<br>`    case v2: c=2; d++;`<br>`      s2`<br>`      /*fall-thru*/`<br>`    case v3: c=3; d++;`<br>`      s3`<br>`      break;`<br>`    default: c=0; d++;`<br>`      s4`<br>`      break;`<br>`  }`<br>`  SAVE(c);`<br>`  SAVE(d);`<br>`}` | `{`<br>`  char c, d;`<br>`  RESTORE(d);`<br>`  RESTORE(c);`<br>`  switch( c )`<br>`  {`<br>`    case 0:`<br>`      rs4`<br>`      if(--d <= 0) break;`<br>`    case 3:`<br>`      rs3`<br>`      if(--d <= 0) break;`<br>`    case 2:`<br>`      rs2`<br>`      if(--d <= 0) break;`<br>`    case 1:`<br>`      rs1`<br>`  }`<br>`}` |

The variable `c` serves to normalize the case values to fall in the range $0..n-1$. Since the case expression is permitted to be of any integral type, the normalization serves to reduce the memory requirements to remember the switch value (assuming the expression is not invariant in the switch body). The variable `d` serves to record the number of case labels that are traversed during the forward execution. This is necessary because **C** permits fall-through execution in the absence of `break` statements between case labels, and since `break` statements could potentially be present inside conditional statements.

Note that the save and restore operations on `c` and `d` are actually generated as part of the transformation phase on compound statements, as described in section 2.2.5.

### 2.2.8 `while` Statement

Each `while` statement is made reversible by using a counter variable to remember the actual number of iterations that were executed. This local variable is saved after the execution, and restored before reversal, as illustrated in the following.

| Original | Transformed | Reversed |
|---|---|---|
| `while( condition )`<br>`    s` | `{`<br>`  int c = 0;`<br>`  while( condition )`<br>`  {`<br>`    c++;`<br>`    s`<br>`  }`<br>`  SAVE(c);`<br>`}` | `{`<br>`  int c;`<br>`  RESTORE(c);`<br>`  while( c > 0 )`<br>`  {`<br>`    rs`<br>`    --c;`<br>`  }`<br>`}` |

It is important to increment the counter `c` *before* executing `s`, since it is possible for `s` to contain jump instructions that take control out of the `while` body. Note that this transformation makes the `while` statement reversible even in the presence of any jump instructions (`break`, `continue`, `goto` and `return`). This is as a result of the combining the normalization of `break`, `continue` and `return` statements (see sections 2.1.8 2.1.7 and 2.1.6) with the transformation of `goto` statements (see sections 2.1.10 and 2.2.5).

Note that the save and restore operations on `c` are actually generated as part of the transformation phase on compound statements, as described in section 2.2.5.

### 2.2.9 `do-while` Statement

Each `do-while` loop is transformed in a manner similar to that for `while` statement, as illustrated in the following.

<table>
<tr><th>Original</th><th>Transformed</th><th>Reversed</th></tr>
<tr><td><pre>do
  s
while( condition );</pre></td><td><pre>{
  int c = 0;
  do
  {
    c++;
    s
  } while( condition );
  SAVE(c);
}</pre></td><td><pre>{
  int c;
  RESTORE(c);
  do
  {
    rs
    --c;
  } while( c > 0 );
}</pre></td></tr>
</table>

## 2.3 Optimization

Several optimizations are possible after the input code is subjected to the normalization and transformation phases of compilation. The more the sophistication of the compiler with respect to the optimizations, the better is the efficiency of the forward and reverse code. Some of the optimizations are described next.

### 2.3.1 Irreversible and Environment Slices

In general, certain operations performed in the input code must be ignored for reversal purposes. There are two reasons for this:

1. Irreversible operations: In any program, certain operations can be assumed to be irreversible for practical purposes. For example, output statements may not be reversed. A message-send over the network cannot always be physically withdrawn.
2. Environment: In certain applications some operations are indeed reversible, but the application has other (probably more efficient) means of reversing those operations. For example, in optimistic parallel simulation, the retraction of a events scheduled by a rolled back event is performed by the parallel simulation kernel. Such operations must be ignored during reverse compilation.

Without loss of generality, both the preceding types of operations can be deemed to be irreversible for the purposes of reverse compilation.

The slices of the program that only lead to such irreversible operations can be *sliced away* during reversal. In other words, for those portions of the forward computation that solely lead to irreversible computation is (1) no attempt is made to make them reversible during forward computation, and (2) such code is ignored during reverse computation. Applying this optimization can result in reduced overheads for reversibility.

**Example:** Consider the following code fragment in which the value of a variable **y** is computed and printed.

| Original | Unoptimized Transform | Unoptimized Reverse | Optimized Transform | Optimized Reverse |
|---|---|---|---|---|
| `{`<br>`int x;`<br>`s1`<br>`{`<br>`int y;`<br>`s2`<br>`y = f(x);`<br>`print(y);`<br>`s3`<br>`}`<br>`s4`<br>`}` | `{`<br>`int x;`<br>`s1`<br>`{`<br>`int y;`<br>`s2`<br>`y = f(x);`<br>`print(y);`<br>`s3`<br>`SAVE(y);`<br>`}`<br>`s4`<br>`SAVE(x);`<br>`}` | `{`<br>`int x;`<br>`RESTORE(x);`<br>`rs4`<br>`{`<br>`int y;`<br>`RESTORE(y);`<br>`rs3`<br>`rs2`<br>`}`<br>`rs1`<br>`}` | `{`<br>`int x;`<br>`s1`<br>`{`<br>`int y;`<br>`s2`<br>`y = f(x);`<br>`print(y);`<br>`s3`<br>`}`<br>`s4`<br>`SAVE(x);`<br>`}` | `{`<br>`int x;`<br>`RESTORE(x);`<br>`rs4`<br>`{`<br>`rs3`<br>`rs2`<br>`}`<br>`rs1`<br>`}` |

The value of `y` is computed as a function of `x`, and printed as output. The slice of the program that only leads to the irreversible printing consists of the variable `y` and the statements `y=f(x);` and `print(y);`. All the components of this slice can be safely eliminated in the reverse code.

### 2.3.2 Eliminating Reversal of Initialization

Even though initializations are destructive assignments, they never need to be undone, and hence don't need to be state saved. This is because initially the variables contain *don't care* values, whose exact values are immaterial for the correctness of the program. For reversal, the same variables can hold any values, which are not necessarily the same *don't care* values that were held before initialization, since any value trivially qualifies as a *don't care* value. For example, consider the following code fragment, and its unoptimized and optimized reverse codes.

| Original | Unoptimized Transform | Unoptimized Reverse | Optimized Transform | Optimized Reverse |
|---|---|---|---|---|
| `{`<br>`int x;`<br>`x = 0;`<br>`s`<br>`}` | `{`<br>`int x;`<br>`SAVE(x);`<br>`x = 0;`<br>`s`<br>`SAVE(x);`<br>`}` | `{`<br>`int x;`<br>`RESTORE(x);`<br>`rs`<br>`RESTORE(x);`<br>`}` | `{`<br>`int x;`<br>`x = 0;`<br>`s`<br>`SAVE(x);`<br>`}` | `{`<br>`int x;`<br>`RESTORE(x);`<br>`rs`<br>`}` |

The unoptimized version saves the pre-initialization value of `x` before initializing it to zero, and restores `x` to the saved value during reverse execution towards the last statement. Since there is no other statement in the reverse code that depends on the pre-initialization value of `x`, such restoration is useless. Hence, both the saving and restoration code for pre-initialization values is eliminated in the optimized version.

This optimization can result in significant memory savings, since local variables are very commonly declared and initialized inside functions.

Note that this optimization is not restricted to local (stack) variables, but also applicable to variables allocated on the heap. The same applies to memory buffers whenever they are re-initialized for irreversible reuse. In fact, a simple generalization of this optimization is the following:

> For memory items that contain *don't care* values, (destructive) assignment of values to them need not be reversed.

### 2.3.3 Invariant Expressions

Determining if an expression is invariant in a block of code is useful in performing certain optimizations to reduce overheads. For example, no condition variable needs to be added for an `if` statement if it is true that the expression is invariant in both branches of the `if` statement (i.e. the condition expression is guaranteed to evaluate to exactly the same value both in forward and in reverse). This fact can be determined by checking the bodies of the true and false branches of the `if` statement to see if any of the variables used in the condition are indeed modifiable in the branches. If none of the variables contained in the condition is affected in the bodies, then the result of the condition need *not* be saved, but instead the exact same condition expression can be re-evaluated to determine the condition value both in forward and reverse codes. In case the condition expression is not invariant in the body of the `if` statement, the result of the condition needs to be saved during the forward execution. However, for this purpose, only one bit is needed to remember the truth value of the condition. Transformations for both the cases (variant and invariant) are described in section 2.2.6.

Similary, for a `switch` statement, an additional variable is introduced to remember the value of the `switch` expression, lest it is changed in the body of the `switch`. This additional variable can be eliminated if the `switch` expression can be determined to always be invariant in the `switch` body.

A simple *sufficient* condition for the the invariance of any expression is if all the variables of the expression are qualified as `const` in their declaration. This trivially ensures that the expression is always invariant.

### 2.3.4 Common Sub-expression Elimination

Even though `rcc` increases the number of sub-expressions and variables in order to render all expressions side effect free, the potential increase of sub-expressions is mitigated by the fact that common sub-expression elimination will later be performed by the usual **C** compiler `cc` when the generated code is compiled.

### 2.3.5 Tape Compression

Run-length encoding and decoding can be performed on the tape. This can result in significant memory savings, for example, in case of loops involving conditional statements. Consider the following code pattern that commonly occurs in programs.

```
done = 0;
while( !done )
{
  if( expr )
  {
    done = 1;
  }
  else
  {
    ...
  }
}
```

In the preceding code fragment, the loop is executed until a condition `expr` becomes true. If the condition expression is modified in the `if` branches, then its truth value (1 bit) must be saved for each iteration (see section 2.3.3). Suppose the loop is executed $n$ times. It is clear that all the first $n-1$ bits will hold a 0 value, and only the last bit will contain a value 1. If run length encoding is used on the tape, then the first $n-1$ bits can be reduced to $O(\lg n)$ bits.

## 2.4 Miscellaneous

In the compilation process, some miscellaneous issues must be considered, as described next.

### 2.4.1 Tape Layout

The order in which data items are saved on the bit/byte tape is important for correctness of reverse execution. The tape layout enables the correct type of information to be available at the appropriate points in the reverse execution. The layout rules are described in the following.

- `if`: The result of the `if` condition must be saved *after* its true or false branch is executed. This is because the number of bits required of either branch is generally unknown in advance, and hence makes it difficult during reverse execution to find out which branch to reverse. Saving the condition value after the branches are executed makes the condition value to become the last item on the tape at the time of reversing the `if` statement, which can be readily popped from the tape.
- `while` and `do-while`: Similar to `if` statement, the counter variables for `while` and `do-while` loop statements must be saved *after* the loops are ended. This ensures that the iteration count is readily available as the last item on the tape at the time of reversing the loop statement.
- `switch`: If the `switch` statement is transformed into another reversible `switch` statement (section 2.2.7), instead of being normalized into `if` and `goto` statements (section 2.1.9), then the flag and fall-through counter variables used in the transformation must be saved *after* the entire body of the switch statement is executed. This enables the counter values to be recovered easily from the end of the tape before the reversal of the switch statement begins.

### 2.4.2 Fault Detection

Reverse execution can be used in trapping errors in either forward code or the reverse code, or even in the implementation of the reverse compiler. This is achieved by checking the following simple *necessary* correctness condition at runtime[2]:

> Suppose a variable `var` is initialized to `expression` in the forward execution. At the end of the reverse execution, the final value of the variable `var` must equal `expression`.

The preceding simple condition, when applied on every local variable and global variable, is useful in detecting errors in the code or the compiler. Another simple correctness is the following, which is a generalization of the end-of-tape condition described in [2]:

> Suppose the bit/byte tape is at position $P$ before the forward execution of a function $f$, and later the reverse of the function is executed. Then, by the end of the end of the reverse execution, the tape must be rewound to the same position $P$.

### 2.4.3 Limitations

A limitation of the source-level reversibility for procedural programs, as presented here, is that the techniques need to be carefully extended and/or modified for other paradigms such as object-oriented programming languages.

For example, consider the reversibility of $C^{++}$ programs. The techniques for reversing **C** programs are trivially applicable to $C^{++}$ programs as well, since all $C^{++}$ programs can be translated to equivalent **C** programs. However, efficiency concerns of such an approach must be resolved carefully. For example, the run-time manipulation of the `this` pointer variable, the handling of virtual function tables, and the maintenance of run time type information, must be investigated to minimize their degeneration into destructive assignments.

Another important open issue is that of efficiently reversing the exception handling mechanisms, such as the `throw` `try` and `catch` primitives of $C^{++}$.

# 3 RCC: Reverse C Compiler

We have implemented the techniques described in previous sections in a reverse **C** compiler called `rcc`. As a basic starting point for the implementation, the **C** parse tree generator from University of Wisconsin, called `CTREE`, has been used. All compilation actions that realize the reversal techniques described in this document have been added over the top of the basic parse tree created by `CTREE`.

The compiler takes a list of file names containing ANSI **C** code as input. For each of the input files, the compiler generates the reversible forward code as well as the reverse code. Additionally, a log file is generated for each file, in which various reverse compilation actions are logged.

For each function `f()` in the input, two functions are generated in the output. The first retains the signature of the input function, and is made reversible. The second function is the inverse of the input function, with the same parameter list, but returning `void`.

Redundant empty block statements and empty (null) statements that may be added during optimization and code generation phases are removed in the final output. The final output is beautified and pretty-printed in human-readable format.

## 3.1 Tape Driver Interface

The compiler generates code assuming a specific interface for bit/byte manipulation, but leaves the implementation of the manipulation to the application. This allows the implementation to be customized in a way that is most efficient for any given application. The application must define the implementation for the following interface for bit/byte tape manipulation. Two different implementations are necessary, one for bits as shown in table 3.1, and the other for bytes as shown in table 3.1.

| **Signature** | **Description** |
|---|---|
| `INIT_BITS()` | Initializes the bit tape |
| `SAVE_BITS(var,n)` | Saves the lowest `n` bits in the integral variable `var` |
| `RESTORE_BITS(var,n)` | Restores the lowest `n` bits into the integral variable `var` |

Table 1: Bit tape interface

| **Signature** | **Description** |
|---|---|
| `INIT_BYTES()` | Initializes the byte tape |
| `SAVE_BYTES(var)` | Saves all the `sizeof(var)` bytes of the *lvalue* `var` |
| `RESTORE_BYTES(var)` | Restores all the `sizeof(var)` bytes of the *lvalue* `var` |

Table 2: Byte tape interface

## 3.2 Pragmas

A set of pragma specifications is available for the user with which certain options can be communicated to the compiler. The list of pragmas is shown in table 3.2. Some of these pragmas

| Pragma | Description | Example |
|---|---|---|
| MAP_REVERSE=$f$,$r$ | Specifies $r$ as the reverse function of $f$. If $r$ is specified as `void`, all calls to $f$ in the forward code will be removed in the reverse code. By default, the reverse of $f$ is `REVERSE_`$f$. | `#pragma MAP_REVERSE=SAVE,RESTORE`<br>`#pragma MAP_REVERSE=printf,void` |
| INVARIANT= CONDITION | Specifies that the condition of the immediately following `if` statement is invariant in both the true and false branches of code. This information is used by the compiler to eliminate the need to store the result of evaluating the condition for realizing reversibility. | `#pragma INVARIANT=CONDITION`<br>`if( ... )`<br>`{`<br>`   ...`<br>`}` |
| INVARIANT= *var-list* | Specifies that given local variables are invariant in the following block scope; this information is used by the compiler to turn off saving the values of the specified variables when they go out of scope. By default, all variables are saved before they go out of scope. This pragma must appear immediately before the compound statement begins. | `#pragma INVARIANT=x,y`<br>`{`<br>`   int w, x, y, z;`<br>`   ...`<br>`}` |
| RECOVERABLE= *var-list* | Similar to INVARIANT. Specifies that the given local variables are recoverable, and hence do not need to be saved before going out of scope. This pragma must appear immediately before the compound statement begins. | `#pragma RECOVERABLE=w,z`<br>`{`<br>`   int w, x, y, z;`<br>`   ...`<br>`}` |
| BEGIN_GLOBAL | Marks pragmas appearing between BEGIN_GLOBAL and END_GLOBAL to be globally applied. | `#pragma BEGIN_GLOBAL`<br>` #pragma MAP_REVERSE=printf,void`<br>` #pragma MAP_REVERSE=max,max`<br>`#pragma END_GLOBAL` |
| STATELESS_SLICE | Marks the immediately following compound statement to consist exclusively of stateless code. This implies state saving can be turned off for destructive assignments, and local variables need not be saved before they go out of scope. However, the reversal of statements, and reverse function calls are performed. | `#pragma STATELESS_SLICE`<br>`{`<br>`   ...`<br>`}` |

Table 3: List of pragma specifications

can be eliminated (or automatically inserted by a smart pre-processor) using sophisticated compiler techniques. The pragmas are only manifestations of the current limitations of the compiler implementation; a production version of the compiler can eliminate the need for such pragmas. For example, the `INVARIANT=CONDITION` pragma for an `if` statement can be eliminated in a sophisticated compiler implementation which can perform advanced dependency analysis between the `if` condition and the `if` branches.

# 4 Application to Parallel Simulation

We used the `rcc` reverse **C** compiler for automatically generating reverse computation corresponding to event computations in the optimistic parallel execution of simulation applications. (In optimistic simulation, synchronization is achieved via rollback of incorrect computations[4].) The optimistic parallel simulator that was used is called the Georgia Tech Time Warp (GTW) system[5]. The details of this application along with a performance study are presented in [7].

In order to use `rcc` in this setting, we provided the event processing functions as input to `rcc`, and obtained their augmented-forward and reverse functions as output, which were then used in the simulation. A tape driver was defined by including the tape data to be part of the event data of a message. Each event processing function `f()` receives exactly one event as input, and modifies a set of variables called state, and can generate additional events during the event processing. In case the event needs to be rolled back, the reverse function of `f()` gets invoked, in order to undo the changes performed by `f()` on the state. The events that were scheduled by that function will be canceled by the GTW kernel. Hence no additional action is needed for reversing the generation of those events.

## 4.1 Event Model Optimizations

The generation of new events as part of an event is a very common and fundamental operation in discrete event models. Two important optimizations can be performed with respect to event generation:

1. **Event scheduling slice**: In optimistic simulation, those events that were generated by an event that is eventually rolled back are retracted by the simulator kernel automatically, using efficient data structures and other simulation techniques[4]. Hence the portion of the event processing code that solely contributes towards the generation of new events is a message-send slice (see section 2.3.1), which can be sliced away in the reverse code. For example, when scheduling new events, a future time-stamp must be specified at which the new event will be scheduled. The time-stamp may be generated using, say, an arithmetic expression involving a random number generator. The time-stamp may in fact be evaluated and stored in a temporary local variable, and then that variable may be used while scheduling the new event. This time-stamp variable, and its computation contribute solely towards the event scheduling, and hence belongs solely to the event-sending slice. Thus, the time-stamp variable and its computation can be safely removed from the reverse code.
2. **Event data initialization slice**: Similar to the event scheduling slice, the event initialization slice involves the initialization of the new event data. All the code that initializes the event data can be safely sliced away in the reverse code, since that need not be reversed.

The elimination of event initialization and scheduling slices together with other optimizations can result in significant reduction in the reverse code overheads.

### Example

Consider the following code fragment extracted from an actual simulation model of a PCS network[6], highly simplified here for illustration.

**Original Model**

```
while( f(x) )
{
  x += g();

  if( h(x) )
  {
    /*schedule
     an event*/
    ...
  }
}
```

In this model, the variable `x` is updated in an loop until a certain value of `x` is reached. As part of the loop, for each value of `x` matching a certain condition, a new event is scheduled.

A straight-forward transformation of this model can result in unoptimized versions of forward and reverse code, as illustrated in the following.

**Unoptimized Forward**

```
int n=0;
while( f(x) )
{
  char c;

  n++;

  x += g();

  if( c = !!h(x) )
  {
    /*same as
      original*/
    ...
  }
  SAVE(c);
}
SAVE(n);
```

**Unoptimized Reverse**

```
int n;
RESTORE(n);
for( ; --n >= 0; )
{
  char c;
  RESTORE(c);
  if( c )
  {
    /*undo event
      scheduling*/
    ...
  }
  x -= rg();
}
```

However, if the event-scheduling slicing is applied in this model, the body of the `if` statement becomes empty in the reverse code, making it a null statement. This removes the need to save the condition value in the forward code, thus removing the reversibility overheads of the `if` statement altogether. The optimized version is as follows.

**Optimized Forward**

```
int n=0;
while( f(x) )
{
  n++;

  x += g();

  if( h(x) )
  {
    /*same as
      original*/
    ...
  }
}
SAVE(n);
```

**Optimized Reverse**

```
int n;
RESTORE(n);
for( ; --n >= 0; )
{
  x -= rg();
}
```

### Reduced Overheads

The unoptimized reverse code requires remembering the result of evaluating the `if` condition. If the loop executes $n$ iterations, then $n$ bits are required for reversibility. In addition, the destructive assignments in the event generation code must be state saved, requiring additional bits. However, in the optimized version, the event scheduling code is eliminated in the reverse, based on the optimization of event scheduling and initialization slicing. This implies that the result of the `if` condition is *never* never saved or restored. Hence, the optimized version requires at least $n$ less bits than the unoptimized version. The scheduling of the event requires event allocation, event data initialization and event scheduling. If these are not eliminated from the reverse, they will need to be state saved in the forward code (which uses several destructive assignments to accomplish the allocation, initialization and scheduling). All such overheads are eliminated in the optimized version.

## 4.2 Tape Driver Implementation

The implementation of the tape driver interface as depicted in Figure 4.2 was used to retarget the rcc compiler to the GTW application.

**Bit driver interface for GTW**

```
void GTW_RESET_BITS( void );
void GTW_SAVE_BITS( int *addr, int nbits );
void GTW_RESTORE_BITS( int *addr, int nbits );

#define INIT_BITS() do{                                            \
     GTW_RESET_BITS();                                             \
  }while(0)
#define SAVE_BITS(_var,_nbits) do{                                 \
    int _x=_var; GTW_SAVE_BITS(&_x, _nbits);                       \
  }while(0)
#define RESTORE_BITS(_var,_nbits) do{                              \
    int _x; GTW_RESTORE_BITS(&_x, _nbits); _var=_x;                \
  }while(0)
```

**Byte driver interface for GTW**

```
void GTW_RESET_BYTES( void );
void GTW_RESET_INTS( void );
void GTW_SAVE_BYTE( char *addr );
void GTW_SAVE_INTS( int *addr, int nbytes );
void GTW_RESTORE_BYTE( char *addr );
void GTW_RESTORE_INTS( int *addr, int nbytes );

#define INIT_BYTES() do{                                           \
    GTW_RESET_BYTES(); GTW_RESET_INTS();                           \
  } while(0)
#define SAVE_BYTES(_var) do{                                       \
    if( sizeof(_var) == 1 ) GTW_SAVE_BYTE(&(_var));                \
    else                    GTW_SAVE_INTS(&(_var), sizeof(_var));  \
  }while(0)
#define RESTORE_BYTES(_var) do{                                    \
    if( sizeof(_var) == 1 ) GTW_RESTORE_BYTE(&(_var));             \
    else                    GTW_RESTORE_INTS(&(_var), sizeof(_var)); \
  }while(0)
```

Figure 1: Tape driver implementation for GTW.

# 5 Summary

A compilation approach to reversibility of **C** programs is presented. The compilation is a three-stage process, involving normalization, transformation and optimization phases. The compilation techniques correctly handle complex data types and control flow, including inter-mixing jumps and recursion. Important optimizations, such as message-slicing, are described that help reduce the overheads that are introduced in the process of making the programs reversible.

The implementation of the reversible compilation techniques in a general-purpose reverse **C** compiler called rcc is described. The rcc compiiler can be used for any application that requires that portions of the application be reversible. It accepts arbitrary **C** functions as input, and, for each input function, it generates (1) reversible version of the input function, and (2) its reverse function. Both the reversible and reverse functions are also expressed in the **C** language.

The compiler can be easily retargeted to any application. As a driving application, the `rcc` compiler has been applied in the domain of optimistic parallel simulation. The compiler is used to generate reversible and reverse code for event computations of simulation models, to support event rollback. The compiler-generated code has been used in a performance study, which confirmed that the generated code performs almost as efficiently as optimized, hand-written code, on sample high-performance simulation applications.

A simple `pragma` interface to the compiler allows the user to specify optimization hints to perform various optimizations. The compiler retains the desirable feature of requiring no modifications to the input **C** source of the model (except for the insertion of `pragma` directives), in order to make the model reversible. In fact, the `pragma` directives serve as a performance enhancing feature; the correctness of the reversible and reverse code is not influenced by the `pragma` hints. Although currently the performance optimization is mostly driven by `pragma`s, the compiler can be enhanced with more sophisticated dependency analysis to automate the optimization process.

# 6 Open Issues

There are several open issues with respect to the reverse computation approach to reversibility as presented here, as compared to the alternative state saving approach described elsewhere, in which every modified variable is saved before modification. Some of them are as follows.

1. How can it be determined if a given function is perfectly reversible (zero state)? What are the necessary and sufficient conditions under which a function is perfectly reversible (taking iteration, recursion and multiple-assignment into account)?
2. If a function is *not* perfectly reversible, how can it be determined whether or not reverse computation requires less state than (incremental) state saving?
3. Are there any data types (even under certain constraints or restrictions) for which reverse computation has provably lower memory complexity than state saving (without sacrificing the time complexity)?
4. Is there a way to prevent the degeneration to incremental state saving in the case of copying the data of an arriving external message into local state?

# References


[1] B. Biswas, R. Mall. "Reverse Execution of Programs." *ACM SIGPLAN Notices*, Vol. 34(4), April 1999.

[2] P. Bishop. "Using Reversible Computing to Achieve Fail-safety." *ISSRE 97, IEEE Computer Society Press*, 1997.

[3] M. A. Ellis, B. Stroustrup. "The Annotated C$^{++}$ Reference Manual." *Addison Wesley Publishers*, 1990.

[4] R. M. Fujimoto. "Parallel Discrete Event Simulation." Communications of the ACM, pages 30–53, vol. 33(10), October 1990.

[5] S. Das, R. M. Fujimoto, K. Panesar, D. Allison and M. Hybinette. "GTW: A Time Warp System for Shared Memory Multiprocessors." *In Proceedings of the 1994 Winter Simulation Conference*, December 1994, pages 1332-1339.

[6] C. D. Carothers, R. M. Fujimoto, and Y-B. Lin. "A Case Study in Simulating PCS Networks Using Time Warp." In *Proceedings of the* $9^{th}$ *Workshop on Parallel and Distributed Simulation (PADS'95*, June 1995, pages 87–94.

[7] C. Carothers, K. S. Perumalla, R. M. Fujimoto. "Efficient Optimistic Parallel Simulations using Reverse Computation." *In Proceedings of 13$^{th}$ Workshop on Parallel and Distributed Simulation*, May 1999.