Efficient Optimistic Parallel Simulations using Reverse Computation

Christopher D. Carothers
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street
Troy, New York U.S.A. 12180-3590
chrisc@cs.rpi.edu

Kalyan S. Perumalla
and
Richard M. Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{kalyan,fujimoto}@cc.gatech.edu

Abstract

In optimistic parallel simulations, state-saving techniques have been traditionally used to realize rollback. In this article, we propose *reverse computation* as an alternative approach, and compare its execution performance against that of state-saving. Using compiler techniques, we describe an approach to automatically generate reversible computations, and to optimize them to transparently reap the performance benefits of reverse computation. For certain fine-grain models, such as queuing network models, we show that reverse computation can yield significant improvement in execution speed coupled with significant reduction in memory utilization, as compared to traditional state-saving. On sample models using reverse computation, we observe as much as three-fold improvement in execution speed over traditional state-saving.

1 Introduction

Parallel simulation approaches can be broadly categorized as optimistic or conservative, depending on whether (transient) incorrect computation is ever permitted to occur during the execution. Optimistic parallel simulations permit potentially incorrect computation to occur, but undo or *roll back* such computation after realizing that it was in fact incorrect. The "computation" in simulation applications is one in which a set of operations, called the *event computation*, modifies a set of memory items, called the *state*. Hence, in order to roll back a computation, it is sufficient to restore the modified memory items to their values before the computation.

The most common technique for realizing rollback is *state-saving*. In this technique, the original value of the state is saved before it is modified by the event computation. Upon rollback, the state is restored by copying back the saved value. An alternative technique for realizing rollback is *reverse computation*. In this technique, rollback is realized by performing the inverses of the individual operations that are executed in the event computation. The system guarantees that the inverse operations recreate the application's state to the same value as before the computation.

To our knowledge, reverse computation has not been previously explored as a viable alternative to traditional state-saving. In this paper, we demonstrate that using reverse computation for realizing rollback can lead to much more efficient executions compared to state-saving. Fine-grain applications (i.e., those with very small amount of computation per event) are examples in which the performance improvement can be most pronounced. This is due to the fact that traditional state-saving operations constitute significant overheads in fine-grain simulations. Also, by reduced memory requirements of the execution, reverse computation leads to more efficient use of storage hierarchies. Reverse computation can greatly reduce the forward computation overheads by transferring most of the traditional overheads to the reverse computation path.

Here, we demonstrate that the reverse computation approach has insignificant forward computation overheads and low state memory requirements in fine-grain models. The parallel simulation performance of reverse-computation is observed to achieve better caching effects, with as much as two to three-fold improvement in several model configurations, compared to copy state-saving. Finally, we demonstrate that this approach can be automated using compiler-based techniques that can automatically generate both a reversible version of the event computation code and its reverse, from a model's high-level description.

When reverse computation is used to simulate coarse-grain models, it is unclear if the improvement in execution speed can be as pronounced, because state-saving overheads are not so high in coarse-grain models. However, coarse-grain models do stand to benefit from reduction in state memory utilization when reverse computation is used.

The reverse computation approach presented here is not meant as a blanket replacement for classic state-saving approaches, but instead to complement or supplement them. Our view is that for many complex applications, no single rollback solution will suffice and that a marriage of this technique and others will be required to yield the most efficient execution of the simulation model.

In Section 2, we present the details of the reverse computation technique using a simple illustrative application, followed in Section 3 by the automation techniques for applying to more complex applications. In Section 4, we present the performance comparison between reverse computation and state-saving. To place our work in context, in Section 5, we identify the work related to general reverse computing in theory and practice. This work opens several interesting challenges and questions, which we identify in Section 6.

2 Reverse Computation

In this section, we illustrate the reverse computation approach with a simple example. For simplicity, we postpone the generalized treatment of more complex models to Section 3.

2.1 Motivating Example: ATM Multiplexor

Consider a simple model in Figure 1 of a non-preemptive ATM multiplexor, containing a buffer of size B. Suppose we are interested in measuring the cell loss probability, and the delay distributions on the queue [3].

```
int qlen;
int sent;
int lost;
int delays[B];
```
(a) state

```
if( qlen < B )
  {
  delays[qlen]++;
  qlen++;
  }
else
  {
  lost++;
  }
```
(b) cell arrival

```
if( qlen > 0 )
  {
  qlen--;
  sent++;
  }
```
(c) cell transfer

Figure 1: A simple ATM multiplexor model.

Then the state of the system might be as shown in Figure 1 (a). The `qlen` variable is used to keep track of the current buffer occupancy; `sent` and `lost` are variables that accumulate the statistics respectively of the total number of cells transferred to the output link and the total number of cells dropped because of a full buffer. The array `delays` measures the number of cells experiencing a given amount of delay, which in combination with the `sent` counter gives the cell delay distribution.

In order to model the behavior of the ATM multiplexor, two types of event handlers are used in the model. The cell arrival event handler, processes newly arriving cells, as shown in Figure 1 (b). Upon a cell arrival, if the queue has no more room, then the counter `lost` is incremented representing that the cell has been dropped. Otherwise, the array element `delay[qlen]` is incremented representing that one more cell experienced a delay of `qlen` emission time units followed by an increment to `qlen` which represents that a cell has been added to the queue. The cell transfer event handler processes cell departure events, as shown in Figure 1 (c). Here, if the queue is not empty, then a cell is dequeued (i.e., `qlen` is decremented) and sent over the output link (i.e., `sent` is incremented).

Note that for both event handlers, the code to schedule the cell arrival and cell departure events is not shown.

2.2 Approach

```
int qlen;
int sent;
int lost;
int delays[B];

bit b1;
bit b2;
```
(a) state

```
if( qlen < B )
  {
  b1 = 1;
  delays[qlen]++;
  qlen++;
  }
else
  {
  b1 = 0;
  lost++;
  }
```
(b) cell arrival

```
if( qlen > 0 )
  {
  b2 = 1;
  qlen--;
  sent++;
  }
else
  {
  b2 = 0;
  }
```
(c) cell transfer

Figure 2: Modified ATM multiplexor model.

Now consider the model shown in Figure 2, which is obtained by slightly modifying the original model of Figure 1. The difference between the two models is that two additional *bit* variables have been added to the state of the original model, and these variables are used to note whether the `if` statements were executed or not. The two bit-variables correspond to the two `if` statements in the model, such that `b1` = 1 if `qlen` $<$ `B` and 0 otherwise. Likewise, `b2` = 1 if `qlen` $>$ 0 and 0 otherwise[1].

If we look carefully at the model, we can see that the *state* of the original model is fully captured by the bit variables `b1` and `b2`. In other words, the state-trajectory of the set S of the variables $\{qlen, sent, lost, delays\}$ has a one-to-one correspondence with that of the set $S' = \{b1, b2\}$. The point here is that the values of the variables in S can be easily recovered based only on the values of S'. To recover, we can run the event computations backwards, which will restore the variables of S to their before-computation values. More abstractly, the bit variables `b1` and `b2` are used to make the original model *reversible*. Indeed, it is easy to find the reverse code for each of the event handlers of the modified model, which is shown in Figure 3. For example, the reverse code shown in Figure 3 (a) performs a perfect undo of the operations of the cell arrival event handler given in Figure 1 (b). Thus, it is sufficient to maintain the history of the bits `b1` and `b2`, instead of the whole set of state variables S of the original model.

```
if( b1 == 1 )
  {
  qlen--;
  delays[qlen]--;
  }
else
  {
  lost--;
  }
```
(a) Reverse cell arrival

```
if( b2 == 1 )
  {
  sent--;
  qlen++;
  }
```
(b) Reverse cell transfer

Figure 3: Reverse code for ATM multiplexor model.

It is clear that the size of the state to be saved is dramatically reduced, from several hundreds of bytes (for S) to just 2 bits (for S'), which can be saved with negligible overhead in the forward computation. As an example, assuming one full word is needed to represent 2 bits on most machines, and if $B = 100$, then the state is reduced by a factor of $(100 + 3)/1 = 103$ when compared to copy state-saving. Even if incremental state-saving techniques are applied to this model, several bytes are needed for saving the changed data values, whereas two bits are sufficient for reverse computation.

2.3 Application Properties

We can make some observations to understand some of the properties of the model that allowed us to reduce the state so dramatically.

- **Property 1** The majority of the operations that modify the state variables are "constructive" in nature. That is, the undo operation for such operations requires no history. Only the most current values of the variables are required to undo the operation. For example, operators such as $++$, $--$, $+=$, $-=$, $*=$ and $/=$ belong to this category[2]. More complex operations such as *circular*

[1] In fact, only one bit variable would be sufficient in this model, since the event handlers are mutually exclusive; but we shall use two variables for clarity in the discussion.

[2] The $*=$ and $/=$ operators require special treatment in the case of multiply or divide by zero, and overflow/underflow conditions.

shift (*swap* being a special case), and random number generation also belong here.

In the multiplexor model, all the assignment operations are constructive. Hence, little extra information is needed to reverse those operations.

- **Property 2** The complexity of the code is such that the "control state" of the code occupies less memory than the "data state" of the variables.

  In the multiplexor model, only two bits were necessary to record the control flow information. In contrast, the data state that is modified is much larger.

If property 1 is not satisfied in the model because of the presence of non-constructive operations such as plain assignment or modulo computation, the reverse computation method can in fact degenerate to the conventional state-saving operations. We call such non-constructive operations as *destructive assignments*. A straight-forward method to reverse a destructive assignment is to save the old contents of the left-hand-side as a record of the "control information" for that assignment statement, which makes it degenerate to state-saving. However, later in the discussion, we shall present optimizations that are possible to prevent the degeneration of destructive assignments to traditional state-saving.

If property 2 is not satisfied because the code is "too complex" (i.e., the amount of control state is more than the data state), we can fall back to traditional state-saving techniques. On the other hand, property 2 suggests that this mechanism is well-suited for simulation models in which the event computations are small.

Queuing network models are an excellent example of the domain of models in which the preceding two properties are satisfied to a large extent. Consequently, we believe that reverse computation is well suited for the optimistic simulation of queuing network models.

3 Automation

In the case of the multiplexor example, the code is small enough for us to come up with its reverse code by inspection. We will now consider the more general case in which the code is complex, requiring a methodical, automated solution for generating the reverse code and for reducing the state size.

3.1 Code Generation

We propose compiler-based techniques to be used to generate the reverse computation code for the simulation model. In our approach, the source code of the original model is fed through a special compiler. From the input model, the compiler generates two separate outputs. The first output is an instrumented version of the input model, which contains the necessary code to make the input code reversible (e.g., the code in Figure 2). The second output is the reversing code that serves to undo the effects of the input model (e.g., the code in Figure 3). In the actual simulation, the instrumented code is used in place of the original code. The reversing code is invoked to rollback an event. The goal of the compiler is to generate the most efficient versions of both the instrumented code and the reverse code such that the state size is minimized while simultaneously reducing the runtime execution overhead.

A simple set of translation rules that can be used by the compiler are shown in Table 1. We list the most common types of statements used in high-level languages, and their corresponding instrumented and reverse code outputs. Against each of the statements, we list the state size achievable for that statement type. Since not all operations of the input model are perfectly reversible, it is necessary to add control state information to be able to reverse them. However, as we shall see in Section 3.3, the better the understanding of the semantics of the code, the better the ability to reduce the state size. Hence, the reduction in state size can vary depending on the sophistication of the compiler. The translation rules of Table 1 thus place an *upper bound* on the state size, which could potentially be improved via optimizations.

The instrumented forward computation code, as well as reverse code, are generated by recursively applying the rules of Table 1 on the input model. The significant parts of these rules are their state bit size requirements, and the reuse of the state bits for mutually exclusive code segments. We explain each of the rules in detail next.

- **T0**: The `if` statement can be reversed by keeping note of which branch is executed in the forward computation. This is done using a single bit variable `b`, which is set to 1 or 0 depending on whether the predicate evaluated to true or false in the forward computation. The reverse code can then use the value of `b` to decide whether to reverse the `if` part or the `else` part when trying to reverse the `if` statement.

  Since the bodies of the `if` part and the `else` part are executed mutually exclusively, the state bits used for one part can also be used for the other part. Hence, the state bit size required for the `if` statement is one plus the larger of the state bit sizes, x_1, of the `if` part and x_2 of the `else` part, i.e., $1 + max(x_1, x_2)$.

- **T1**: Similar to the simple `if` statement (**T0**), an `n`-way `if` statement can be handled using a variable `b` of size $lg(n)$ bits. Thus, the state size of the entire `if` statement is $lg(n)$ for `b`, plus the largest of the state bit sizes, $x_1 \dots x_n$, of the component bodies, i.e., $lg(n)+max(x_1 \dots x_n)$ (since the component bodies are mutually exclusive).

- **T2**: Consider an n iteration loop, such as a `for` statement, whose body requires x state bits for reversibility. Then n instances of the x bits can be used to keep track of the n instances of invocations of the body, giving a total of $n * x$ bit requirement for the loop statement. The inverse of the body is invoked n times in order to reverse the loop statement.

- **T3**: A loop with variable number of iterations, such as a `while` statement, can be treated the same as a fixed iteration loop, but the actual number of iterations executed can be noted at runtime in a variable `b`. The state bits for the body can be allocated based on an upper limit n on the number of iterations. Thus, the total state size added for this statement is $lg(n) + n * x$.

- **T4**: For a function call, no instrumentation is added. For reversing it, its inverse is invoked. The inverse is easily generated using the rules for **T7** described later. The state bit size, x, is the same as for **T7**. Treatment of recursive functions is discussed later in this section.

- **T5**: Constructive assignments, such as ++, −−, += and so on, do not need any instrumentation. The reverse code uses the inverse operator, such as −−, ++, −= respectively. These constructive statements do not require any state bits for reversibility.

- **T6**: Each destructive assignment, such as =, `%=` and so on, can be instrumented to save a copy of its left hand side into a variable `b` before the assignment takes place.

The size of `b` is $8k$ bits for assignment to a k-byte left hand side variable (*lvalue*). This is similar to Steiman's incremental state-saving technique [13].

- **T7**: In a sequence of statements, each statement is instrumented depending on its type, using the previous rules. For the reverse code, the sequence is reversed, and each statement is replaced by its inverse, again using the corresponding generation rules from the preceding list. The state bit size for the entire sequence is the sum of the bit sizes of each statement in the sequence.
- **T8**: Jump instructions are discussed later in this section.
- **T9**: Any legal nesting of the previous types of statements can be treated by recursively applying the corresponding generation rules. The state bit size is also obtained by the corresponding state-bit composition rule.

Jump Instructions

Jump instructions (such as `goto`, `break` and `continue`) require more complex treatment, especially in the presence of inter-dependent jumps. As a simple example, consider the case in which no `goto` label in the model is reached more than once during an event computation. Such use of jump instructions occurs, for example, to jump out of a deeply nested `if` statement, or as convenient error handling code at the end of a function. Such models are easy to reverse, as follows: for every label that is the target of one or more `goto` statements, its `goto` statements are indexed. The forward code is instrumented to record the index of a `goto` statement whenever that `goto` statement is executed. In the reverse code, each of the `goto` statements is replaced by a `goto` label. The original (forward) `goto` label is replaced with a switch statement that uses the index saved in forward computation to jump back to the corresponding new (reverse) goto label. Since at most one index per `goto` label is stored, the bit size requirement of this scheme is $lg(n)$ where n is the number of `goto` statements that are the sources of that single target label. Note that even if a label is the target of only one jump instruction, at least one bit is required, to distinguish between reaching the label normally (falling-through) and reaching the label as a result of the jump instruction.

In the most general case of models containing arbitrarily complex use of jump instructions, we believe it is difficult to fully optimize the state requirements automatically. Hence, they are beyond the scope of this initial research in the area of reverse computation and will require further investigation.

Functions and Recursion

Function calls also need complex treatment, especially in the presence of reuse and/or recursion. The complexity is due to the overriding concern of minimizing the forward computation and memory overhead while trying to maintain reversibility. In the simple case in which the function call graph is a tree, the state bit sizes can be completely determined *statically*, and hence the state bits can be *statically* allocated to the statements in *all* the functions. This can result in efficient references to the state bits with minimal indirection. In the case of models in which the function call graph is a directed acyclic graph (DAG), the (maximum) state bit size requirements can still be statically determined, but the references to the state bits, both in the forward and reverse event computation, need indirection via a *frame offset* variable generated by the compiler. The *frame offset* denotes the position in the bit vector from where a forward function can begin storing its own reversibility state. This variable is analogous to a frame pointer in a function call stack. In the more general case of an arbitrary function call graph (implying the presence of direct and/or indirect recursion), it is difficult to statically determine the maximum state bit sizes. However, the *frame offset* approach of DAGs can still be used to refer to the state bits corresponding to the currently active function invocation. The actual run-time performance implications of these techniques is unclear, and requires detailed study. The analogy to function call stack also points to potential relationship of this approach to process-oriented simulations, which requires further investigation as well.

State Size Determination

To determine the amount of state needed to reverse an event computation, the following procedure is used. Since the model code is a sequence of statements, start with T7 (or, alternatively, T4), and recursively apply the rules of Table 1. This is done while reusing the bits on code segments that are mutually exclusive (as indicated by the MAX() operation in the table). The analogy of *register allocation* is applicable here. The state can be seen as a sequence of bits, which correspond to registers of a computer. The bits are allocated to the state that is required to record control-flow information. Just like registers, these bits can be allocated in an intelligent manner so that mutually exclusive statements can reuse the same bits. For registers in general computing, the savings are in computation time; for control bits in optimistic simulations, the savings are in state copying operations and in state size reduction.

It is easily observed that the statements with potentially higher state bit sizes are destructive assignments, nestings of conditional statements within loops, nested loops inside loops, and destructive operations among inter-dependent jump instructions. In fine-grain models, it is unlikely that complex code involving nested or complex loops will arise. Hence, the higher state requirements of such complex code is not a serious problem for these models. However, destructive assignments are not uncommon. The most common occurrence of destructive assignments is in random number generation, which is addressed in the next section, followed by a discussion on other efficiency issues in achieving reversibility.

3.2 Reversible Random Number Generation

Random number generation is central to all simulations models. Several random number streams may be used in the same simulation, to model various phenomena. A random number stream is generated by repeatedly invoking a specified function on a *seed* variable. The function modifies the value of the seed every time the function is invoked. Thus, a seed variable is needed for every random number stream used in the simulation model. The size of the seed variable varies with the type and quality of the random number generator (RNG).

In optimistic simulations, if an event computation invokes an RNG, and eventually the event is rolled back, it is necessary to roll back the random number generation. Otherwise, the simulation results can be unpredictable and unrepeatable. In order to be able to roll back the random number generation, traditionally, the seed value is state-saved. Incremental state-saving techniques are used in case the model contains many seeds.

However, if the reverse computation approach is used in order to avoid state-saving, we quickly encounter the following problem — RNGs rely on lossy/destructive assignments such as modulo operations. This implies that a straightforward application of reverse computation techniques can degenerate to incremental state-saving, as the generation rule for type **T6** in Table 1 suggests. To get around this problem, we essentially need RNGs which do not rely on state-saving to reverse. On an abstract level, we can reasonably expect RNGs to be *reversible* without the need for state-saving, since, after all, random number streams are nothing but statically laid out

Table 1: Summary of treatment of various statement types

Generation rules and upper bounds on state size requirements for supporting reverse computation. s, or $s_1..s_n$ are any of the statements of types T0..T7. inv(s) is the corresponding reverse code of the statement s. b is the corresponding state-saved bits "belonging" to the given statement. The operator $=@$ is the inverse operator of a constructive operator $@=$, (e.g., $-=$ for $+=$).

Type	Description	Application Code			Bit Requirements		
		Original	Instrumented	Reverse	Self	Child	Total
T0	simple choice	if() s_1; else s_2;	if() { s_1; b=1;} else { s_2; b=0;}	if(b==1) {inv(s_1);} else {inv(s_2);}	1	x_1, x_2	$1 + max(x_1, x_2)$
T1	compound choice (n-way)	if() s_1; elsif() s_2; elsif() s_3; else() s_n;	if() { s_1; b=1;} elsif() { s_2; b=2;} elsif() { s_3; b=3;} else { s_n; b=n;}	if(b==1) {inv(s_1);} elsif(b==2) {inv(s_2);} elsif(b==3) {inv(s_3);} else {inv(s_n);}	$lg(n)$	$x_1, x_2, \ldots, x_n$	$lg(n) + max(x_1, \ldots x_n)$
T2	fixed iterations (n)	for(n) s;	for(n) s;	for(n) inv(s);	0	x	$n * x$
T3	variable iterations (maximum n)	while() s;	b=0; while() {s; b++;}	for(b) inv(s);	$lg(n)$	x	$lg(n) + n * x$
T4	function call	foo();	foo();	inv(foo)();	0	x	x
T5	constructive assignment	$v@=w;$	$v@=w;$	$v=@w;$	0	0	0
T6	k-byte destructive assignment	$v=w;$	$\{b=v; v=w;\}$	$v=b;$	$8k$	0	$8k$

T7	sequence	s_1;
		s_2;
		s_n;
			s_1;
			s_2;
			s_n;
				inv(s_n);
				inv(s_2);
				inv(s_1);
					0	$x_1 + \ldots + x_n$	$x_1 + \ldots + x_n$

T8	simple case of jumps (label lbl as target of n goto's)	goto lbl; s_1;
		goto lbl; s_n;
		lbl: s;
			b=1; goto lbl; s_1;
			b=n; goto lbl; s_n;
			b=0; label: s;
				inv(s);
				switch(b) {
				case 1: goto $label_1$;
				case n: goto $label_n$; }
				inv(s_n); $label_n$:
				inv(s_1); $label_1$:
					$lg(n)$	0	$lg(n)$

T9	Nestings of T0-T8	Apply the above recursively			Apply the above recursively		

cyclic sequences of numbers. It should be possible to traverse forward and backward along the cycles with the same ease.

More concretely, consider the code to generate a uniform random number using L'Ecuyer's Combined Linear Congruential RNG [9]. This RNG is based on a combination of four linear congruential generators (LCGs) and has a period of 2^{121}. This generator produces a uniform $[0, 1]$ `double`. Here, s represents the seed of an LCG. When trying to "undo" or reverse this computation as suggested in Section 3, we immediately run into several destructive assignments. In particular, this generator performs the following assignment:

$$s = 45991 * (s - k * 46693) - k * 25884$$

where $k = s/46693$. Because integer division is being used (and in fact the algorithm depends on the semantics of integer division), k does not accurately represent $s/46693$ which means that one cannot determine the original value of s from the new value of s. Essentially, there is loss of information, making it irreversible. Using the step-wise technique of reversing a computation, the only way one could reproduce the original value of s from the previous value is to store the loss of information due to the integer division (and due to other operations like it) and use that information in the reverse computation. However, this degenerates to state-saving, which is exactly what we are trying to avoid.

Now, let us examine the mathematics behind this RNG from a higher level. This RNG is based on the following recurrence:

$$x_{i,n} = a_i x_{i,n-1} \, mod \, m_i$$

where $x_{i,n} | 1 \leq i \leq 4$ is n^{th} set of four seed values computed from the $n-1$ set of four seed values, $m_i | 1 \leq i \leq 4$ are the primes numbers $2^{31} - 2, 2^{31} - 106, 2^{31} - 226, 2^{31} - 326$ respectively, and $a_i | 1 \leq i \leq 4$ is a primitive root for m_i. Based on well-known number theory, the above recurrence form is in fact reversible. First, the inverse of a_i of modulo m_i, b_i is defined to be:

$$b_i = a_i^{m_i - 2} \, mod \, m_i$$

where calculation of b_i is accomplished using the method for computing large powers [17]. Using the b_i, we can generate the reverse sequence of seed values as follows:

$$x_{i,n-1} = b_i x_{i,n} \, mod \, m_i$$

which has the same computational requirements as the forward execution of the RNG.

Significance of Reversible RNG

The reversibility of RNGs is not new. However, when applied to the context of parallel simulation, the work described here is the first to exploit this property. As the gap between memory latency and processor speed increases, we believe this approach will be of greater benefit, as faster processors will result in larger, more complex simulation applications. These simulation applications will in turn require RNGs with stronger statistical properties and longer periods, which will increase the seed size of the RNG. For example, in [10], the "Mersenne Twister" (MT19937) RNG is presented. This RNG is of the twisted feedback shift-register class and has an extremely long period of $2^{19937} - 1$. However, it requires 624 words of space for seeds. For a classical Time Warp system using this generator, 2496 bytes of state would need to be saved per event just to support the "undo" operation for the RNG. This assumes MT19937 would be called at least once per event. One might think that incremental state-saving could be employed here, but the way this RNG is structured, some bits from each word are subject to change every time a random number is generated, thus making it difficult to optimize using incremental state-saving techniques. Assuming the reverse recurrence can be found for MT19937, which its creators believe is possible, the amount of memory saved using reversing computation is even much greater than previously discussed. Because of the reduction in state-saving overheads, system performance will improve as well.

3.3 Reverse Code Efficiency

The reversibility of random number generators, even though they contain destructive assignments, leads to the following third property of the models that can help prevent reverse computation from degenerating to state-saving:

- **Property 3:** The non-reversibility of the individual steps that compose a computation do not necessarily imply that the computation, when taken as a whole, is not reversible.

Property 3 suggests that even if the individual steps of a computation are not efficiently reversible (i.e., either property 1 or 2 is violated), then one should look to a higher-level to see if the computation is not reversible from that level. An interesting question we plan to consider in the future is defining an automatic mechanism for identifying code sequences which are individually not reversible, but for which a reversible code sequence can be determined when considered in a larger context.

This observation holds for several other common operations that contain destructive assignments. For example, a shift operation on an array of n elements can require n state-saving operations using incremental state-saving techniques. The same operation requires saving only one element using reverse computation. In fact, a circular shift requires no state when reverse computation is used, whereas incremental state saving can require n state-saving operations[3]. Similarly, insertion or deletion operations (which contain destructive assignments such as pointer assignments) into tree data structures (e.g., priority queues) can require several state-saving operations using incremental state-saving, whereas, no state is needed when reverse computation is used. This is because those operations naturally possess perfect inverses (e.g., *delete* and *insert* are inverses of each other).

An important outcome of this work is the recognition that reverse computation is well-suited for queuing network models. Many of the operations in queuing network models are either constructive operations (increment, decrement, etc.), or reversible groups of destructive assignments (random number generators, queue operations, etc.). Also, the event computations in these models tend to be of fine-granularity. This implies that reverse computation is an excellent approach for optimistic parallel simulation of queuing network models.

4 Performance Evaluation

We shall now discuss the implementation details of reverse computation, followed by our performance study that compares reverse computation against state-saving.

4.1 Implementation

To experiment with reversing computation, we have implemented the reverse computation in the Georgia Tech Time Warp (GTW) optimistic parallel simulator for shared memory multiprocessors. GTW is originally based on state-saving to realize rollback. To use reverse computation for rollback, three significant modifications were made to the GTW kernel.

First, we extended the GTW application programmer interface to support a method for reversing the forward processing of an event. In GTW, the applications programmer must specify methods (i.e., pointer to a function) for each logical process (LP) to (i) initialize an LP (`TWLP[i].IProc`) (ii) primary event handler for an LP (`TWLP[i].Proc`), (iii) a "wrap-up" method for an LP that collects application-specific statistics (`TWLP[i].FProc`). Note that the `TWLP` array is indexed by the LP number. We added support for reversing computation by introducing a new method, `TWLP[i].RevProc`, which performs the precise reverse computation of the event handler procedure, `TWLP[i].Proc`. Currently, the specification of `TWLP[i].RevProc` must be done by the applications programmer. (In the future, we plan to support the automatic generation of `TWLP[i].RevProc` based on `TWLP[i].Proc`, as described in Section 3). The arguments to `TWLP[i].RevProc` include the current state of the LP, and any events sent during the forward computation.

Next, GTW's core rollback mechanism required some significant changes as well. GTW uses a technique called *direct cancellation* [6] to support the "de-scheduling" of previously scheduled events by an event that was rolled back. This technique allows one to keep a direct pointer to the event that needs to be canceled. Because of this, an optimized rollback mechanism can be supported that doesn't require one to search though the processed event-list of an LP. Instead, if the event that is to be canceled has been processed, the rollback mechanism simply restores the version of LP state that was made prior to processing this event. The other processed events that come after the canceled event are marked as unprocessed and placed back into pending event-list. For supporting reversing computations caused by secondary rollbacks (i.e., rollbacks caused by event cancellations), this optimized technique is unsuitable. To "undo" a sequence of event computations using reverse computation requires that each event be "unprocessed" in the precise reverse order in which it was processed. Consequently, we modified the direct cancellation mechanism so that it starts with the last event processed by the LP and moves through the LP's processed event-list in reverse time stamp order, invoking the `TWLP[i].RevProc` method for each event to undo its changes to state. The changes to the primary rollback mechanism (i.e., rollbacks caused by straggler events) to incorporate reverse computation were straight-forward, since the processed event list for an LP is scanned in reverse time stamp order.

The last major change to the GTW system was that all memory allocation for saving state (both copy state and incremental state-saving) was turned off. Also, the copy-state operation during forward event processing was turned off as well. Instead, a small bit vector (currently, a 32-bit integer variable) was added to every event. These bits serve as the working bits needed for saving the state information created by the instrumented model code, as described in Section 3. For example, the two bits, `b1` and `b2` of the multiplexor model in Section 2.2 are in fact mapped to the lower order bits of this event bit vector. To support more complex code, the size of the bit vector can be appropriately increased.

4.2 Experiment Configuration

For the performance study, we use two applications: (i) a network of Asynchronous Transfer Mode (ATM) multiplexors (ii) a Personal Communications Services (PCS) network.

The first application consists of a 3-level cascade of ATM multiplexors, as described in [11]. The model is parameterized by a factor n, such that n^3 cell sources feed into n^2 multiplexors which in turn feed into n multiplexors, which finally feed into one multiplexor. The factor n is the number of inputs of each multiplexor. The GTW source code for the ATM multiplexor model was obtained from the Northern Parallel Simulator (Nops) group at Dartmouth [11]. Their implementation on GTW realizes each network element as an LP. The state size of each LP is 112 bytes. The data contain within each message is 8 bytes. The event granularity of this application is very low: approximately 3 microseconds.

In the second application, a PCS network is simulated as described in [4]. The service area of the network is populated

[3] The commonly used swap operation is only a special case of circular shift.

with a set of geographically distributed transmitters and receivers called *radio ports*. A set of radio channels are assigned to each radio port, and the user in the *coverage area* sends and receives phone calls using the radio channels. When a user moves from one cell to another during a phone call a *hand-off* is said to occur. In this case the PCS network attempts to allocate a radio channel in the new cell to allow the phone call connection to continue. If all channels in the new cell are busy, then the phone call is forced to terminate. For all experiments here, the *portable-initiated* PCS model was used, which discounts *busy-lines* in the overall call blocking statistics. Here, *cells* are modeled as LPs and PCS subscribers are modeled as messages that travel among LPs. PCS subscribers can travel in one of 4 directions: north, south, east or west. The selection of direction is based on a uniform distribution. The state size for this application is 80 bytes with a message size of 40 bytes and the minimum lookahead for this model is *zero* due to the exponential distribution being used to compute call inter-arrivals, call completion and mobility.

The computation granularity of ATM multiplexor model is very small, but, the communication among the LPs is feed-forward in nature, yielding excellent lookahead properties. The PCS network, on the other hand, possesses more complex communication patterns with much larger message sizes and a *zero* lookahead. Consequently, PCS is a more representative example of how a "real-world" simulation model would exercise the rollback dynamics of reverse computation.

To adapt the models to use our reversing computation methodology, we manually generated[4] the reverse event processing code according to the rules discussed in Section 3 (portions of the code are discussed in Section 2.2).

All the experiments were performed on a 16 processor, SGI Origin2000, shared-memory multiprocessor, with 8 MB of level-two cache per processor, and 4 GB of total memory. In all cases, the total number of events committed using reverse computation were deterministic and consistent with state-saving and sequential runs.

To study various performance effects, we used four different versions of GTW as described below:

1. **GTW-SS** – parallel version of GTW, using state-saving
2. **GTW-RC** – parallel version of GTW, using reverse computation
3. **GTW-NONE** – parallel version of GTW, with rollback support turned off (i.e., with neither state-saving nor reverse computation)
4. **GTW-SEQ** – optimized sequential simulator with GTW interface.

4.3 Forward computation

In practice, one would like the serial performance of the parallel simulator to be as close to the optimized sequential as possible. With that in mind, our first set of experiments uses the ATM multiplexor model and compares the serial performance of GTW-NONE[5], GTW-RC, and GTW-SS (using copy state-saving) against GTW-SEQ on this model to determine the impact these different approaches have on forward computation rates. We did not use incremental state-saving in this comparison since it resulted in slower performance than full copy saving-saving. The cause of low performance of incremental state-saving was a consequence of the LP state being so small (only 112 bytes)[7]. We did not use the PCS network model, since it is of a higher granularity than the ATM multiplexor model, and hence less stringent than the ATM multiplexor model on the forward computation overheads.

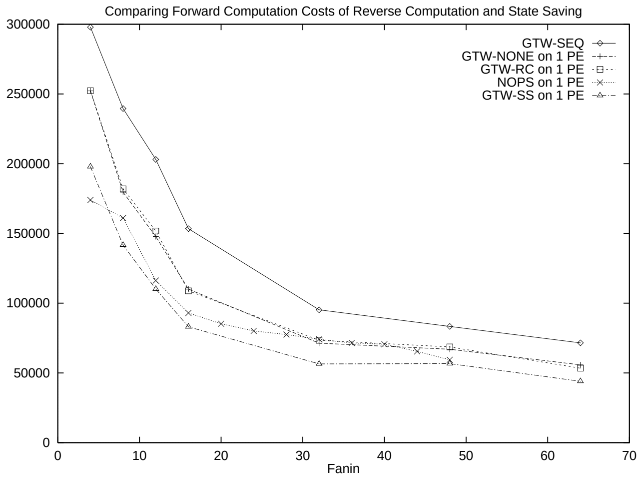

Figure 4: Comparison of forward computation performance to determine overheads in state-saving and reverse-computation approaches using the ATM multiplexor.

Figure 4 shows the event rate as a function of fan-in for the four simulators. There are several key observations based on this performance data. First, we observe that the performance of GTW-RC is equal to GTW-NONE. The reason these two systems perform equivalently is because the few extra bits stored in the forward computation to support reverse computation has negligible impact on the overall event granularity of the ATM Multiplexor application. However, if we compare GTW-RC with GTW-SS, a much different picture emerges — GTW-RC is consistently faster than GTW-SS, the primary reason being that we have completely eliminated the overhead of state-saving.

If one were to eliminate state-saving overheads in an optimistic simulator, as we achieved in GTW-RC, we may expect to observe performance that is about equal to that of the optimized sequential simulator. But, clearly that is not the case here — across all fan-in values, the sequential simulator is faster, and, in one case, as much as 30% faster. To investigate this phenomenon, we profiled GTW-RC and GTW-SEQ to see where these two systems were spending most of their CPU cycles. Profiling revealed that the memory footprint of GTW-RC is much larger than that of GTW-SEQ. This is because the sequential simulator commits and immediately reuses an event memory buffer upon processing that event. But, GTW-RC (and GTW-SS) only commits an event memory buffer when global virtual time (GVT) sweeps past the event time-stamp, which is approximately once every 1000 events. The consequence of waiting for GVT is that GTW-RC "touches" more pages of memory than GTW-SEQ, which results in more first and secondary data cache misses, as well as translation look-aside buffer (TLB) misses and page faults.

Finally, we observe that as the fan-in increases, the performance of the different simulators begins to converge. To explain this phenomenon, we need to understand how an increase in fan-in effects the system. Recall, there are n^3 sources in the multiplexor network. Each source generates two messages — one for self rescheduling, and the other when a cell is generated to send to the target multiplexor. Consequently, there

[4] In the future, we plan to automate this process by modifying the TeD compiler [15] to generate the reverse event handler methods based on the original forward event handler.

[5] GTW-NONE is very much like a conservative parallel simulator being run serially.

are at anyone instance in time at least n^3 events in the system. Thus, the event population grows as the cube of the fan-in, n. As we approach fan-ins of 48 and above, the event-list management overheads begin to dominate, which decreases the impact state-saving overheads has on overall system performance.

To illustrate that the performance of GTW-RC is comparable to that of an optimized conservative simulator which is supplied with sufficient lookahead, in Figure 4 we list the performance of the Nops[11] conservative parallel simulator on the same model on a similar SGI Origin2000 (fan-in beyond 48 are not listed in [11]). We notice that GTW-RC achieves similar or better performance compared to the conservative simulator.

In summary, in the fine-grained multiplexor model, we observe that reverse computation almost completely eliminates the state-saving overheads from the forward computation.

4.4 ATM Multiplexor Parallel Performance

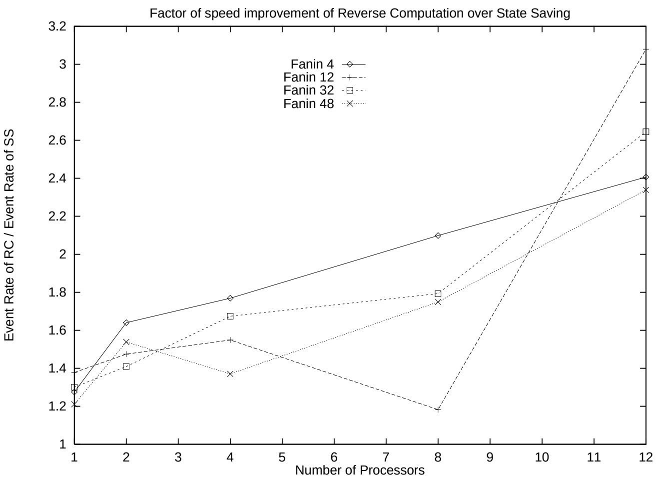

Figure 5: Improvement in parallel simulation performance using reverse-computation as compared to state-saving.

In this next series of experiments, we compare the *parallel* simulation performance achieved by reversing computation and state-saving using the ATM multiplexor model. For these experiments we varied both the fan-in (fan-ins of 4, 12, 32, and 48) and the number of processors (2, 4, 8, and 12). Given the modest, nevertheless good, improvement in serial performance when using reverse computation (around 25%), we expected to see a similar modest enhancement with respect to parallel simulation performance. However, we were surprised to see that reverse computation improved GTW's performance by up to 300% as compared to state-staving. Figure 5 shows the *factor* of improvement (the event-rate of reverse computation divided by the event-rate of state-saving). We observe that in the 12 processor, 12 fan-in case, GTW-RC increased the event-rate by a factor of 3.1 compared to GTW-SS[6]. All the performance data were obtained by repeating the simulation runs several times. The performance results were found to be repeatable, with negligible variance.

[6]The raw event-rate using reverse computation for that case was over 1.2 million events per second!

Table 2: ATM Multiplexor Model, Fan-in 12

Memory Subsystem Performance for Reverse Computation and State-Saving on 12 Processors. Numbers represent *misses* incurred by different parts of the memory hierarchy.

	TLB	PD-Cache	SD-Cache
SS	43966018	1283032615	162449694
RC	11595326	590555715	94771426

These observations raised the next question, namely, why does reverse computation improve performance by such a large factor? We hypothesized that it is memory system related, assuming that reverse computation has a smaller memory footprint than state-saving and hence requires less resources to be expended by the memory subsystem. To verify our hypothesis and to precisely identify the source of the performance variation, we used the `speedshop` performance tool. Here, we configured `speedshop` to make use of the hardware counters internal to the MIPS R10000 processor to obtain extremely accurate performance statistics. We note that because the hardware counters were used, we observe neither slow down in performance, nor perturbation in the model performance due to the `speedshop` monitoring software.

Table 2 shows the number of TLB, primary data cache (PD-Cache), and secondary data cache (SD-Cache) misses for GTW with state-saving and GTW with reverse computation. These measures were obtained for the 12 processor, 12 fan-in case. Here, we observe GTW-RC only incurs 26% of the TLB misses incurred by GTW-SS. This accounts for much of the performance gains of reverse computation, since TLB misses are expensive to service, as they are typically done in software. In addition to the TLB miss statistics, we observed that reverse computation has 50% less primary data cache misses and 40% less secondary data cache misses than state-saving, which further explains the large increase in performance obtained by reverse computation. As future generations of processors become faster and the performance gap between memory and processors widens, we anticipate reverse computation can achieve even higher performance compared to state-saving.

Last, we observe a significant degree of variation among some of the performance results. The largest amount of variation occurs in the cases corresponding to a fan-in of 12 . Here, we see that on 8 processors, reverse computation achieves an event rate that is only 20% faster than that of state-saving. Yet, on 12 processors, we observe an astonishing 300% increase in the event-rate when using reverse computation. We attribute these variations to load imbalance and dramatic changes in inter–processor communication that are inherent in mapping the ATM multiplexor model to different processors.

4.5 PCS Network Parallel Performance

We also simulated the PCS model in parallel, and compared the parallel performance of GTW-SS and GTW-RC. For these experiments the following configurations were used. The PCS model was configured with a 64 x 64 LP grid for 8 processors, a 72 x 72 LP grid for 12 processors, and a 60 x 60 LP grid on 15 processors. For all LP configurations, the number of initial events per LP was 25. These LP configurations were chosen because they allowed an even number of LPs to mapped to each processor to preclude introducing a unbalanced workload.

The performance results for this set of experiments are summarized in Table 3. Here, we observe that GTW-RC is

Table 3: PCS Network Model

Parallel Performance for Reverse Computation and State-Saving. Performance is measured in terms of event rate.

# PEs	GTW-SS	GTW-RC	% Improvement
8	222507	336869	151%
12	199084	510772	256%
15	183292	678380	370%

consistently faster than GTW-SS. The peak performance improvement by GTW-RC is 370% over that of GTW-SS. We believe that, for processor configurations above eight, GTW's state-saving mechanism is being subjected to high TLB miss rates and other memory performance effects, probably related related to the Origin 2000's unique CCNUMA architecture. Further investigation is needed to determine the precise cause. Accordingly, we conservatively believe that the 150% improvement, as reported in the 8 processor case, is more in line with what can be expected for large processor configurations in practice, since some optimizations could potentially be made in GTW-SS for better state-memory and event-memory management in such large processor configurations.

Performance Summary

The results presented here when considered in their total indicate that the performance of optimistic parallel simulation has reached an acceptable level for this class of extremely low event granularity applications. Previously, researchers in the area of parallel and distributed simulation have indicated difficulty in achieving acceptable levels of performances from Time Warp systems with small event-granularity. They observed that state-saving costs were dominating and stifling performance. Now, with reverse computation it appears that arguments against using optimistic approaches on such applications are ebbing away.

5 Related Work

Reverse computation has been previously studied in various contexts. Research into *reversible computing* is aimed at realizing reversible versions of conventional computations in order to reduce the power consumption [1, 14]. The R language is a high-level language with special constructs to enforce reversibility so that programs written in that language can be translated to machine code of reversible computers [5]. Another interesting application of reversible computation is in garbage collection. The Psi-Lisp language presented in [2] uses reversible constructs to efficiently implement garbage collection. Other applications for reversible execution are in the areas of database transaction support, debugging support and checkpointing for high-availability software [12, 8]. More recent work is concerned with source to source translation of popular high-level languages, such as C, to realize reversible programs. However, almost all of the solutions suggested in these application areas translate either to constraints on language semantics to disallow irreversible computations, or to techniques analogous to state-saving techniques (specifically, copy-on-write techniques) of optimistic parallel simulations. Some of them operate at a coarse level of virtual memory pages. The optimizations are roughly analogous to those used in incremental state-saving approaches in parallel simulations. Moreover, since these solutions are not specifically geared towards parallel simulations, they are not optimized for minimizing the state size, and do not adequately exploit the semantics of constructive operations. The state-saving techniques presented in [7] utilize a limited form of optimization using the reverse computation approach and is the first work we are aware of to specifically discuss reverse computation, but no performance results are provided. Our work starts where [7] ends and is concerned with techniques for minimizing the state size for realizing reversibility, and simultaneously minimizing the runtime execution overheads. Finally, in [16], a *rollback relaxation* scheme is presented that automatically identifies certain types of history-independent logical processes and optimizes the performance of rollback activity for those processes. Our approach is different in that it addresses logical processes which are not necessarily stateless, and seeks to optimize run-time performance and memory utilization by minimizing the essential state required by such processes.

6 Remarks and Conclusions

Reverse computation is well suited for models containing constructive assignments. However, it can degenerate to traditional state-saving if sufficiently large number of destructive assignments which are hard to reverse are present in the model. In fact, in certain cases, it can perform worse than incremental state-saving, due to the fact that optimizations, such as the merging of multiple writes to the same variable into a single save operation, are possible using incremental state-saving techniques, but not readily possible with reverse computation.

An optimization that is commonly implemented in copy state-saving is that, when a rollback spans several processed events, it is sufficient to merely switch a few pointers in order to restore the entire state to its value corresponding to the earliest rolled back event. This helps in considerably reducing the rollback cost. In contrast, when reverse computation is used, each one of the rolled back events must be reversed one at at time, in the reverse order of processing. This can potentially make the rollback cost much higher than that of copy state-saving.

On the other hand, previously, optimistic simulations were considered to be unsuitable for fine-grain applications because of the high state saving overheads. We have shown that reverse computation is an appealing alternative approach that makes efficient optimistic simulation of fine-grain applications feasible.

We also identify some classes of applications in which application of reverse computation is natural, for which automatic techniques are easily found, which essentially exploit the source code as state. Examples include quantum computer simulation, and queuing network simulation. In the case of queuing network models, we identify that a majority of the common operations are indeed reversible. In particular, we have addressed the reversibility of the most common operation, namely, random number generation. In addition, we make the observation that other queue manipulation operations, such as insert, delete and shift, are in fact more memory efficient with reverse computation than with state-saving.

In other classes of applications, this approach also serves as an automatic compiler-based state-compression technique. State compression is useful for enhancing the performance of optimistic simulations in limited memory environments. Considering that CPU resources are cheaper and more abundant than memory resources, we can hope to execute certain important classes of applications (such as queuing networks) using optimistic parallel simulation on a network of, say, palm-top computers. The state-compression is useful even in the context of state-logging conservative parallel simulations and sequential simulations. For interactive (play-log-replay) applications, there can be significant benefits in terms of reduction in memory requirements of the state log. Since the applications tend to be simulated for long times, an order of magnitude difference in the size can be quite significant. (In this case, we are still investigating the gains of state-compression as opposed to using standard compression programs, such as `gzip`, on the log of regular uncompressed state.)

Most importantly, the reduced memory requirements due to state-compression allow us to explore new applications that were considered too expensive to simulate using state-saving-based optimistic simulations. However, several open issues remain to be explored. We discuss a few of them next.

Open Issues

In general, reverse computation reduces the overhead in the forward computation path, but increases the rollback cost. Additional work is needed to better understand the rollback dynamics of reverse computation on a wider range of applications.

Algorithms to automatically identify the naturally reversible patterns in the model code are important to prevent reverse computation from degenerating to state-saving. Perhaps a library of forward–inverse pairs of functions can help in this direction.

Since floating point arithmetic is subject to roundoff, arithmetic operations can result in roundoff errors during the reverse execution. Solution approaches exist (for example, by emulating a precision that is higher than the highest precision supported by the modeling language), but the performance implications are unclear.

An interesting theoretical problem is to find whether there exist data types, for which the state-saving cost for their operations widely differs when reverse computation is used instead of state-saving. To illustrate, consider a circular shift operation on an array of n elements. This operation requires no state for reverse computation. But it appears to require $O(n)$ state size using state-saving, if a `for` loop is used for shifting. However, by using a pointer–based implementation for the array, and shifting the "start" and "end" pointers of the array instead of the actual elements, the pointers can be state-saved instead of the entire array of elements, reducing the size of saved state to the size of two pointers. This implies that for circular shift, the memory requirement for state-saving is only a constant factor away from reverse computation. It is unclear if this is true in general. For example, an interesting sub-problem concerns the `insert` and `delete-min` operations on a priority queue. We are not aware of any theoretical result that proves or disproves that only a constant number of *state modifications* is sufficient for arbitrary combination of `insert` and `delete-min` operations on the queue, without sacrificing the asymptotic average time complexity of $O(\log n)$ for insertion and deletion. Reverse computation, on the other hand, requires no state history despite state modifications, because, `insert` can be reversed using `delete`, and vice versa.

Acknowledgments

This work was supported in part by U.S. Army Contract DASG60-95-C-0103 funded by the Ballistic Missile Defense Organization, and in part by DARPA Contract N66001-96-C-8530. Additionally, the authors would like to thank P. L'Ecuyer for his insights on the reversibility of random number generators as well as David Nicol and the Nops group at Dartmouth for providing us with the source code for the ATM Multiplexor model specifically written for GTW. And finally, we would like to thank the anonymous reviewers and shepherds for their insightful suggestions.

References

[1] Charles Bennett. Thermodynamics of Computation. International Journal of Physics, pages 905–940, volume 21, 1982.

[2] H. Baker. NReversal of Fortune—The Thermodynamics of Garbage Collection. In *Proceedings of the International Workshop on Memory Management*, Springer Verlag Lecture Notes in Computer Science 637, 1992.

[3] K. S. Perumalla, C. A. Cooper, R. M. Fujimoto. An Efficiency Prediction Method for ATM Multiplexers. In *Proceedings of Broadband Communications*, April 1996.

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

[5] Michael Frank. The R Language Programming Language and Compiler,
`http://www.ai.mit.edu/~mpf/rc/home.html`

[6] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 3, pages 242–249, August 1989.

[7] Fabian Gomes. Optimizing Incremental State-Saving and Restoration. Ph.D. thesis, Dept. of Computer Science, University of Calgary, 1996.

[8] G. Leeman. A Formal Approach to Undo Operations in Programming Languages. ACM TOPLAS, pages 50–87, volume 8(1), Jan 1986.

[9] P. L'Ecuyer and T. H. Andres A Random Number Generator Based on the Combination of Four LCGs *Mathematics and Computers in Simulation*, 44:99–107, 1997.

[10] M. Matsumoto and T. Nishimura Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 8, Number 1, pages 3–30, January 1998.

[11] A. Poplawski and D. M. Nicol Nops: A Conservative Parallel Simulation Engine for TeD *In Proceedings of the* 12^{th} *Workshop on Parallel and Distributed Simulation*, volume 23, pages 180–187, May 1998.

[12] R. Sosic. History Cache: Hardware Support for Reverse Execution. Computer Architecture News, pages 11–18, volume 22,5, December 1994.

[13] Jeff S. Steinman. Incremental state saving in SPEEDES using C++. *In Proceedings of the 1993 Winter Simulation Conference*, December 1993, pages 687–696.

[14] The Reversible Computing Home Page at MIT,
`http://www.ai.mit.edu/~cvieri/reversible.html`

[15] The TeD Language — Home page.
`http://www.cc.gatech.edu/computing/pads/ted.html`

[16] K. Umamageswaran, K. Subramani, P. A. Wilsey, P. Alexander. Formal Verification and Empirical Analysis of Rollback Relaxation. The Elsevier Science *Journal of Systems Architecture*, number 44, pages 473–495, 1998.p

[17] C. Vanden Eynden Elementary Number Theory page 141, Random House, New York, 1987.