Efficient Optimistic Parallel Simulations Using Reverse Computation

Christopher D. Carothers
Rensselaer Polytechnic Institute
and
Kalyan S. Perumalla and Richard M. Fujimoto
Georgia Institute of Technology

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 six-fold improvement in execution speed over traditional state-saving.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Shared Memory; C.1.2 [**Process Architectures**]: Multiprocessors; I.6.1 [**Simulation and Modeling**]: Types of Simulation—*discrete-event, parallel*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: reverse computation, parallel discrete event simulation, state-saving, rollback

A preliminary version of this article appeared in the 13^{th} *Workshop on Parallel and Distributed Simulation (PADS '99).*

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.

Name: Christopher D. Carothers
Address: Department of Computer Science, 110 8^{th} Street, Troy, New York 12180-3590, `e-mail: chrisc@cc.gatech.edu`
Affiliation: Rensselaer Polytechnic Institute

Name: Kalyan S. Perumalla and Richard M. Fujimoto
Address: College of Computing, 801 Atlantic Drive, Atlanta, Georgia 30332-280, `e-mail: {kalyan,fujimoto}@cc.gatech.edu`
Affiliation: Georgia Institute of Technology

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 speedup in several model configurations when compared to copy state-saving, periodic state-saving and incremental 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 still 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. In order to place our work in context, in Section 5, we identify 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 until 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 [Perumalla et al. 1996].

```
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

Fig. 1. A simple ATM multiplexor model.

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 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

Fig. 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

[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.

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

Fig. 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 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.

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

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 *destructive assignments*. A straightforward 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 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 roll back 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. We have implemented a reverse **C** compiler called `rcc` that realizes these upper bounds for the **C** language[Perumalla and Fujimoto 1999].

The instrumented forward computation code, as well as reverse code, are generated by recursively applying the rules of Table 1 to 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, $x_1 \ldots x_n$, of the component bodies, i.e., $lg(n) + max(x_1 \ldots x_n)$ (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 $lg(n) + n * x$. If an upper limit on the number of iterations is unknown, alternative approaches can be used, such as described in [Perumalla and Fujimoto 1999].

—**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**.
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. 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.

—**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 Steinman's incremental state-saving technique [Steinman 1993].

—**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 (such as `goto`, `break` and `continue`) can be treated in different ways, depending on whether or not inter-mixing sets of jumps are present in the code. In the simple case, 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 $lg(n+1)$ 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.
The more general case of models containing arbitrarily complex use of jump instructions is treated in [Perumalla and Fujimoto 1999].

—**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.

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 simulation 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 straight-forward 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 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 [L'Ecuyer and Andres 1997]. 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.

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	jump (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);
				$label_1$:
					$lg(n+1)$	0	$lg(n+1)$

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

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 an apparent 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 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 [Vanden Eynden 1987]. 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 together will increase the seed size of the RNG. For example, in [Matsumoto and Nishimura 1998], 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 the definition of 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 (the commonly used swap operation is only a special case of circular shift). 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

In order to study the performance of reverse computation relative to state-saving, we compare two flavors of reverse computation with three variants of state-saving. All the variants have been implemented using the Georgia Tech Time Warp (GTW) optimistic parallel simulator for shared memory multiprocessors[Das et al. 1994]:

—**GTW-RC**: GTW with reverse computation, in which the reverse code for the application models is generated manually and optimized by inspection

—**GTW-RCC**: GTW with reverse computation, in which the reverse code for the application models is automatically generated using a special reverse compiler

—**GTW-CSS**: GTW with copy state-saving, in which a copy of the entire state is made before every event

—**GTW-PSS**: GTW with periodic state-saving, in which a copy of the entire state is made every p^{th} event

—**GTW-ISS**: GTW with incremental state-saving, in which a copy of only the modified portions of the state is made during every event.

In addition, the following two sequential versions of GTW are used for comparison purposes:

—**GTW-SEQ**: Optimized sequential simulator with GTW interface

—**GTW-NONE**: Parallel version of GTW, with rollback support turned off (i.e., with neither state-saving nor reverse computation), so that it can be run sequentially, but not in parallel.

We first present the details of these variants, followed by the details of our study to compare their performance characteristics. 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 was deterministic and consistent with sequential runs, and the performance was found to be repeatable.

4.1 Reverse Computation

We have implemented the reverse computation in GTW, which 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`. 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* [Fujimoto 1989] to support the "descheduling" 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 was added to every event, which served 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. The application can declare and use additional bits by specifying them as application-specific event data.

4.2 Reverse Code Generation

In order to use the reverse computation support in GTW, it is necessary to define the reverse function for every application function that is invoked during event processing. To this end, first, we manually wrote the reverse functions by inspection, following the rules in Table 1 (the applications are described later in this section). We will refer to this manual configuration for reverse computation as **GTW-RC**. Next, we used a reverse **C** compiler called `rcc`, which we implemented to automatically generate reverse functions from **C** functions[Perumalla and Fujimoto 1999]. We will refer to this automated configuration for reverse computation as **GTW-RCC**.

4.3 State-Saving

The default state-saving technique in GTW is copy state-saving, in which a copy of the entire state is made before an event is executed. In CSS, the state is saved every time an event is processed. A variation of copy state-saving is called periodic state saving[Fujimoto 1990]. Periodic state-saving is a generalized technique in which state is saved only periodically, say, every p^{th} event, instead of every event as is done with copy state-saving. This implies that some events save state before processing, and others do not. The former set of events can be rolled back easily by restoring the state to the saved values. The latter set of events need special treatment, since they do not have saved state. The state restoration for these events is achieved by starting with a past processed event that does have saved state, and then re-executing the sequence of events from that past event to the event just before the rolled back event.

We incorporated periodic state-saving into GTW as a generalization of copy state-saving. The application can choose between copy and periodic state-saving by specifying its state-saving period to be equal to or greater than unity, respectively. The implementation is optimized for copy state-saving when the period equals unity. No source code changes are necessary in the application models to switch between copy and periodic state-saving. We shall refer to the copy and periodic state-saving configurations as **GTW-CSS** and **GTW-PSS** respectively.

Incremental state-saving is an alternative state-saving technique in which only the modified portions are saved just before modification. GTW includes an implementation of incremental state-saving in which the modifications are logged as pairs of integral address-value pairs, and stored in a log array for each LP. We shall refer to this state-saving configuration as **GTW-ISS**. Although GTW allows both copy state-saving and incremental state-saving to be used simultaneously together in the same application, we used them mutually exclusively, because of the uniformly small state sizes in our applications.

4.4 Applications

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

—**ATM Multiplexor Cascade**: The first application consists of a 3-level cascade of ATM multiplexors, as described in [Poplawski and Nicol 1998]. 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 [Poplawski and Nicol 1998]. Their implementation on GTW realizes each network element as an LP. The state size of each LP is 112 bytes. The application data contained within each message is 8 bytes. The event granularity of this application is very low (a few microseconds for small n).

—**PCS Network**: In the second application, a PCS network is simulated as described in [Carothers et al. 1995]. The service area of the network is populated 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 the 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, has medium event granularity and 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.

4.5 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[3], GTW-RC, and GTW-CSS 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)[Gomes 1996]. 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.

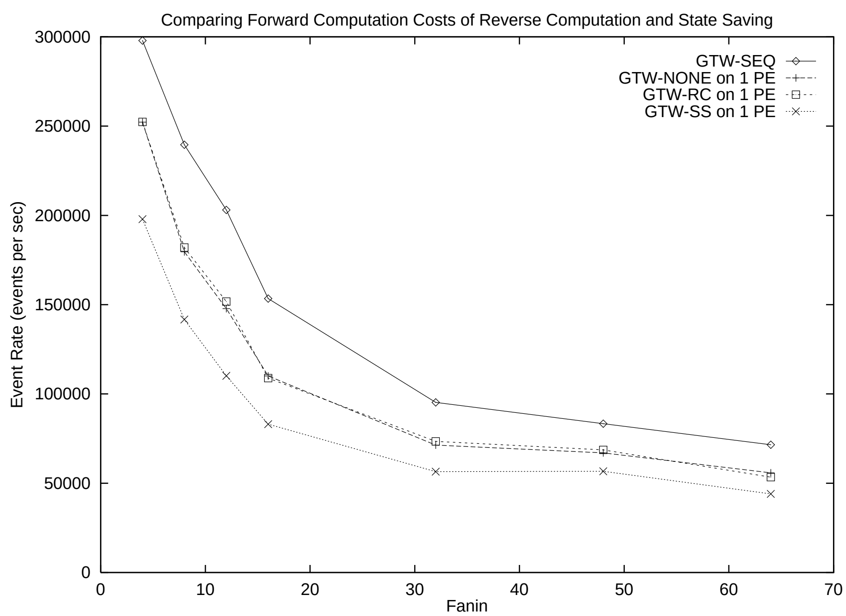

Fig. 4. Comparison of forward computation performance to determine overheads in state-saving and reverse computation approaches using the ATM multiplexor.

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

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

GTW-RC with GTW-CSS, a much different picture emerges — GTW-RC is consistently faster than GTW-CSS, 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-CSS) 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 are, at any one instance, 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 overhead has on overall system performance.

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

4.6 Parallel Simulation Performance

In this next series of experiments, we compare the *parallel* simulation performance achieved by reverse computation and state-saving on the ATM multiplexor model and on the PCS network model.

—**ATM Multiplexor**: For the experiments with the ATM multiplexor model, we chose a representative fan-in of 16 (to get a non-trivial network size that still keeps the event granularity sufficiently small) and varied the number of processors (2, 4, 8, 12 and 16). The experiments were performed separately for each of the GTW configurations – GTW-RC, GTW-RCC, GTW-CSS, GTW-PSS and GTW-ISS. Figure 5 compares the event rate obtained with all the configurations, on a varying number of processors.

—**PCS Network**: We also simulated the PCS model in parallel, and compared the parallel performance of state-saving and reverse computation, using the different GTW configurations on a varying number of processors. For these experiments the following PCS network settings were used. The PCS model was configured with a 64x64 LP grid for 8 processors, a 72x72 LP grid for 12 processors, and a

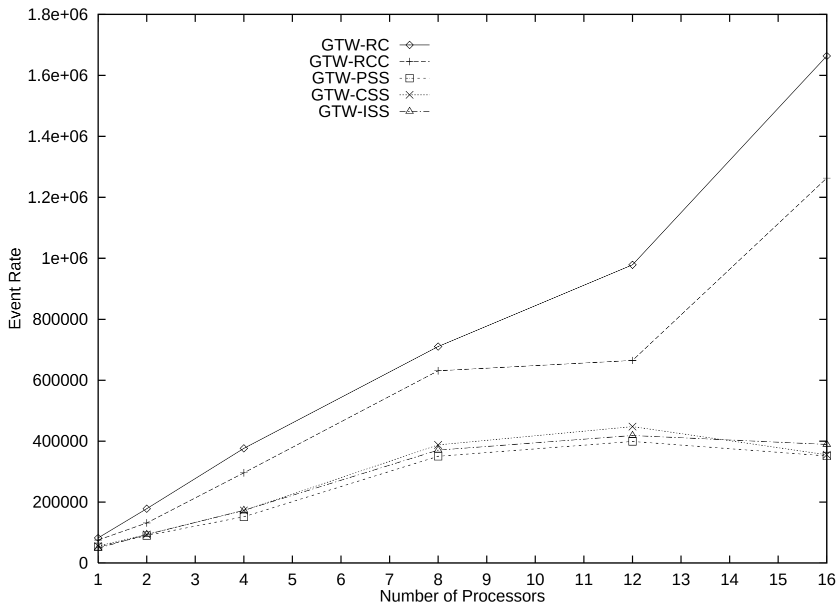

Fig. 5. Event Rate: reverse computation vs. state-saving on the ATM Multiplexor model with fan-in 16 (4,369 LPs).

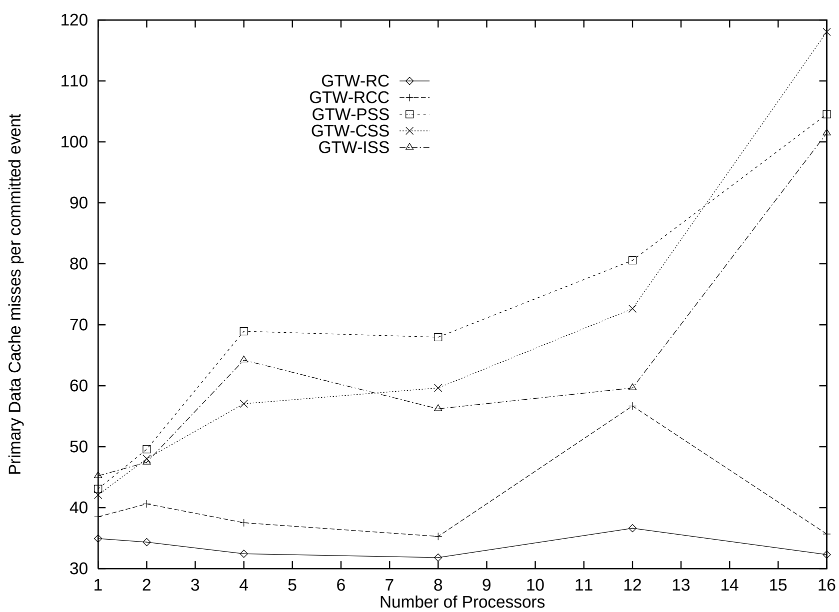

Fig. 6. Primary Data Cache Miss Rate: reverse computation vs. state-saving on the ATM Multiplexor model with fan-in 16 (4,369 LPs).

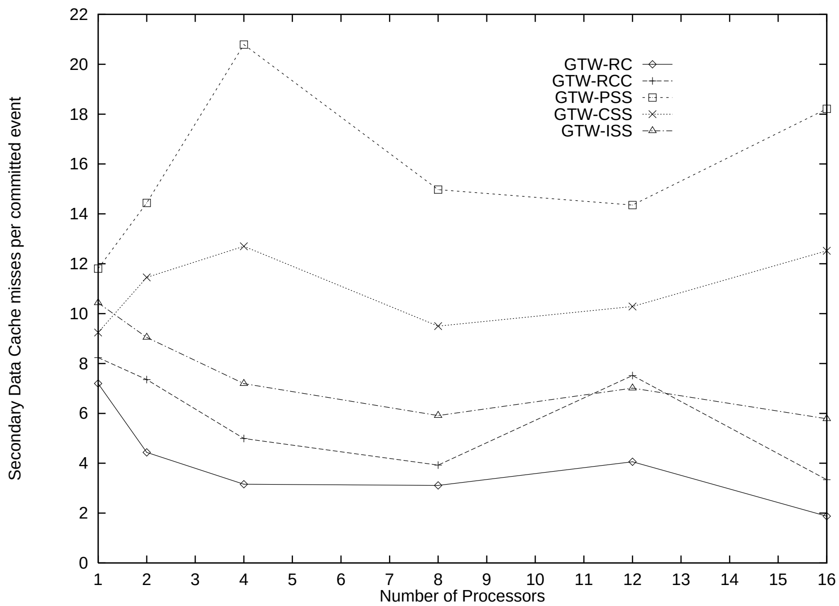

Fig. 7. Secondary Data Cache Miss Rate: reverse computation vs. state-saving on the ATM Multiplexor model with fan-in 16 (4,369 LPs).

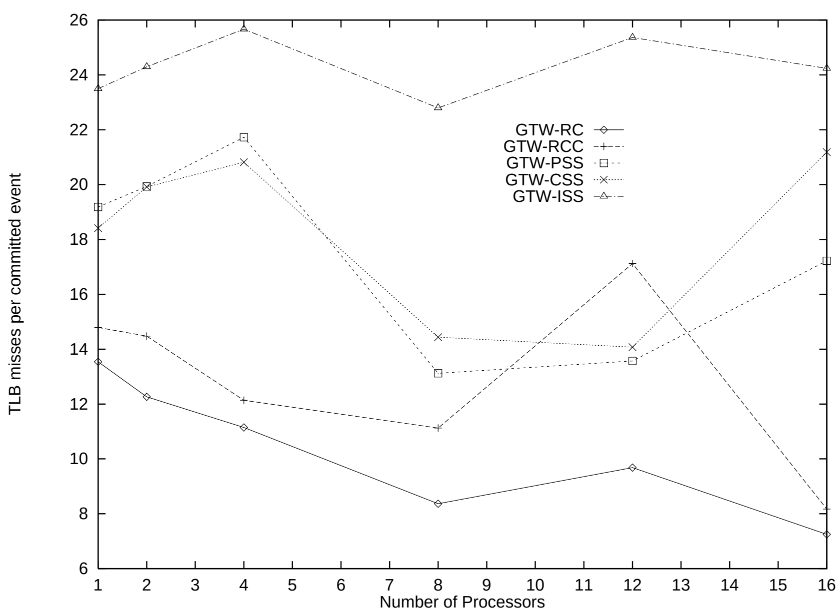

Fig. 8. TLB Miss Rate: rerse computation vs. state-saving on the ATM Multiplexor model with fan-in 16 (4,369 LPs).

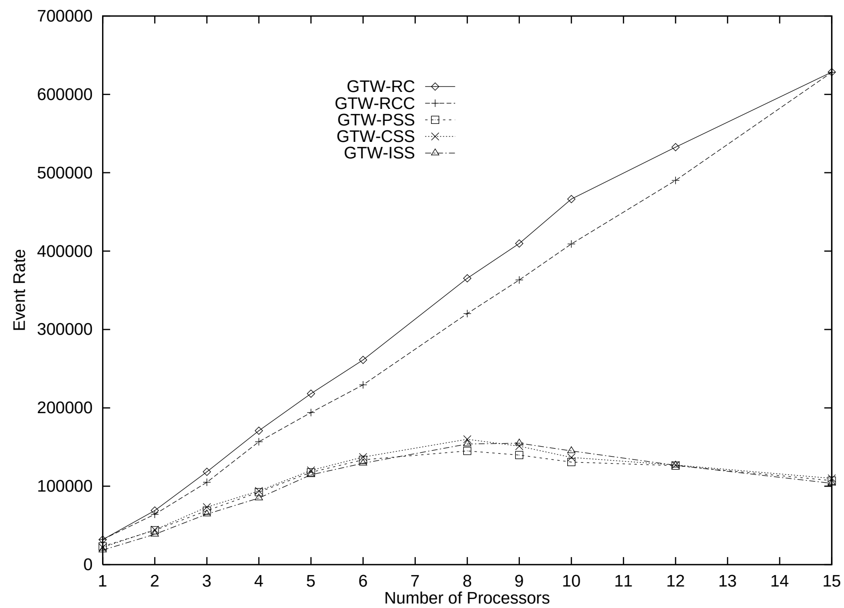

Fig. 9. Event Rate: reverse computation vs. state-saving on the PCS network model with 40,000 LPs.

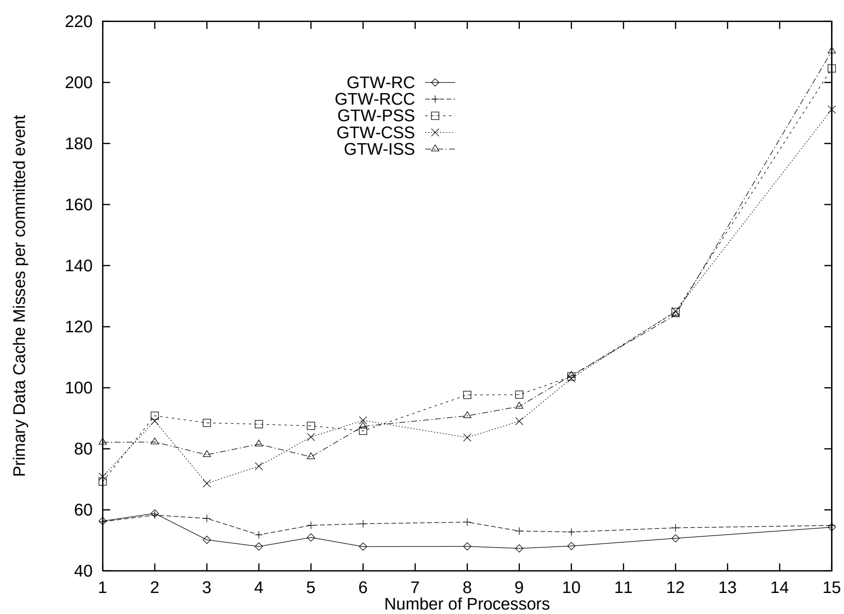

Fig. 10. Primary Data Cache Miss Rate: reverse computation vs. state-saving on the PCS network model with 40,000 LPs.

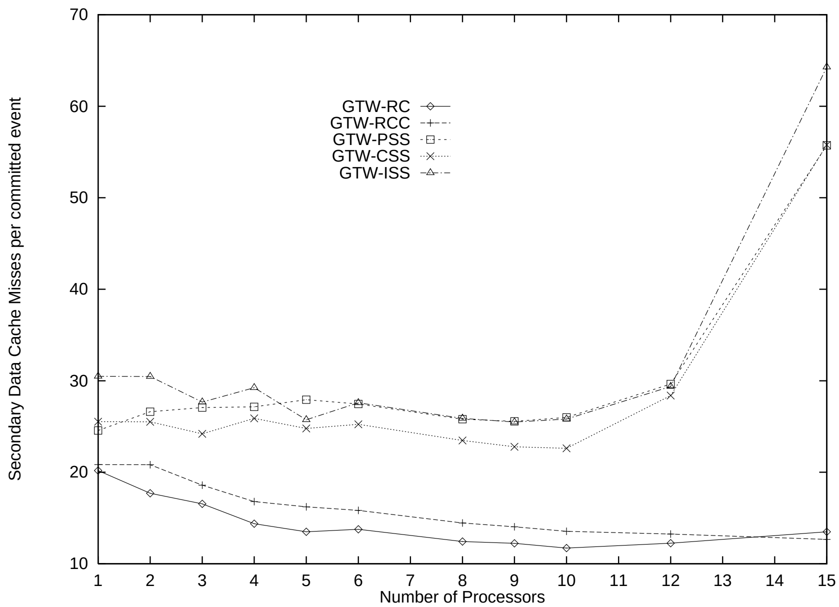

Fig. 11. Secondary Data Cache Miss Rate: reverse computation vs. state-saving on the PCS network model with 40,000 LPs.

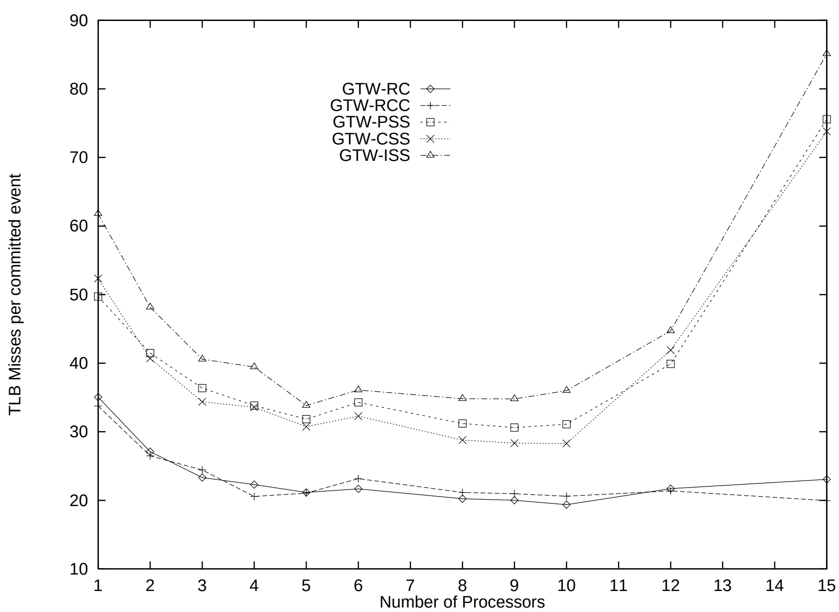

Fig. 12. TLB Miss Rate: reverse computation vs. state-saving on the PCS network model with 40,000 LPs.

60x60 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 be mapped to each processor to preclude introducing an unbalanced workload. The event rate performance for this set of experiments is shown in Figure 9.

Given the modest, nevertheless good, improvement in serial performance when using reverse computation, 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 on the ATM Multiplexor model by up to 300% as compared to state-staving, and up to 500% on the PCS network model. We observe that in the 16 processor case of the ATM Multiplexor model, GTW-RC increased the event rate by a factor of more than 4 compared to GTW-CSS[4]. Similarly, in the 15 processor case of the PCS network model, GTW-RC increased the event rate by a factor of more than 5 compared to GTW-CSS. All the performance data were obtained by repeating the simulation runs several times. The performance results were found to be repeatable, with negligible variance. In all cases, we observe that GTW-RC is consistently and significantly faster than GTW-CSS, GTW-PSS and GTW-ISS.

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 fewer 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 `perfex` performance tool. Here, we configured `perfex` 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 observed neither slow down in performance, nor perturbation in the model performance due to the `perfex` monitoring software.

Figures 6 – 8 show the primary data cache misses, secondary data cache misses and TLB misses for the ATM Multiplexor model. Figures 9 – 12 show the corresponding statistics for the PCS network model. In these figures, the number of misses is normalized by a constant, which is the total number of events committed by the simulation. We observe that GTW-RC incurs significantly fewer number of primary and secondary data cache misses and TLB misses per event, compared to GTW-CSS, GTW-PSS and GTW-ISS. The net effect of the poor memory subsystem behavior of the state-saving variants is that the event rate degrades as the number of processors is increased. Some of the factors behind the low performance of state saving are explained next. More detailed analysis of the detrimental effect of state-saving on the simulation performance on shared memory multiprocessors is presented in [Carothers et al. 1999].

Copy State-Saving. When copy state-saving is used, the footprint of the simulation is increased because of the additional memory required for state maintenance for each event. The increase in the additional memory size manifests itself in terms

[4]The raw event rate using reverse computation for that case was over 1.6 million events per second!

of an increase in the number of TLB misses incurred by the simulation per event. Furthermore, the simulation touches more memory pages per event, due to the act of making a copy of the state. This further increases the chance of cache misses and TLB misses. The misses contribute to contention at the shared memory, with the net effect of rapidly reducing the performance as the number of processors increases. These phenomena are indicated by the rapid increase in the number of primary data cache misses in the ATM Multiplexor simulation, as shown in Figure 6. Similarly, in the case of PCS network simulation, the number of primary and secondary data cache misses, along with the number of TLB misses steadily increase as the number of processors increases, as shown in Figures 10 – 12. The net effect is that the event rate of the simulation deteriorates as more processors are added to the simulation.

Periodic State-Saving. Since periodic state-saving avoids saving state too often, it is fair to expect that its performance would be better than that of copy state-saving. This is because periodic state-saving can potentially reduce the overhead during the forward computation, and reduce the memory consumed for state-saving. This is especially true in applications in which the state size is significantly greater than the event size[Bellenot 1992; Press and MacIntyre 1992]. However, in applications–such as used in this study–in which event size is comparable to (or greater than) state size, periodic state-saving can have the counter-effect of actually increasing the memory utilization relative to copy state-saving. This is attributable to the fact that events cannot be reclaimed until global virtual time (GVT) goes past the earliest among all logical processes of their latest state-saved event. With copy state-saving, events can be reclaimed as soon as GVT sweeps past their time stamps, which allows them to be reused immediately thereafter. In contrast, with periodic state-saving, a processed event can be reclaimed only if there exists another processed event with saved state whose timestamp is less than GVT. Hence, the working set of simulations using periodic state-saving is greater in size relative to that of copy state-saving, due to a greater number of events maintained between GVT computations. (In fact, this relation is preserved independent of the frequency with which GVT is computed.). The net result is that periodic state saving does not help in significantly reducing the state-saving overheads relative to copy state-saving. This is confirmed by the similarity of the memory subsystem performance of GTW-CSS and GTW-PSS, as shown in Figures 6 – 8, and in Figures 10 – 12.

Incremental State-Saving. Since incremental state-saving avoids copying the entire state, and copies instead only the pieces that have been modified, it can normally be expected to incur less overhead than copy state-saving. This is because, incremental state-saving can potentially consume less memory than copy state-saving in applications with large state sizes. If only a small portion of the state is modified per event, then incremental state-saving can result in significant reduction in state-saving overheads per event relative to copy state-saving. However, in applications–such as used in this study–in which the state size is small, the overheads of maintaining a log of changes (lists of address-value pairs) is significant, making it no better than copy state-saving. This is confirmed by the similarity of the memory subsystem performance of GTW-CSS and GTW-ISS, as shown in Figures 6 – 8 and in Figures 10 – 12.

Automated Reverse Code. Finally, we note that the hand-coded reverse code (GTW-RC) performs slightly better than the automated reverse code (GTW-RCC). This is because the hand-coded reverse code incorporates more optimizations than the automatically generated reverse code. For example, bit operations are more customized for each application in the hand-coded version, whereas the automated version uses more generalized bit operations. Consequently, the automated version adds a slight amount of additional computational overhead over that of hand-coded version. This is evident in the slightly reduced event rates for GTW-RCC relative to GTW-RC, as shown in Figures 5 and 9. The fact that the memory characteristics of GTW-RC and GTW-RCC remain the same is indicated by the similarity of their memory performance results, as shown in Figures 6 – 8 and in Figures 10 – 12.

4.7 Performance Summary

The results presented here, when considered in their totality, 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 (e.g., [Xiao et al. 1999]). 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.

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.

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 power consumption [Bennet 1982; MIT Reversible Computing Group 1999]. 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 [Frank 1999]. Another interesting application of reversible computation is in garbage collection. The `Psi-Lisp` language presented in [Baker 1992] 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 [Leeman 1986; Sosic 1994; Biswas and Mall 1999]. 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.

In [Bishop 1997], reversible computing has been suggested as a method for testing failures in real-time systems, but with admittedly high forward and reverse computing overheads, and without treatment of complex instructions such as intermixing jumps. An initial attempt at automatically generating symbolic inverses of *reversible* functions is made in [Eppstein 1985], but it relies on heuristics for correctness. A more theoretical approach is taken in [Chen and Udding 1990], by using inversion of invariants to prove the correctness of inverse programs. A debugging system is described in [Biswas and Mall 1999] that executes **C** programs in interpreted mode in forward and reverse directions. Although their approach using interpretation is well suited for debugging systems, the performance characteristics of their techniques are unclear when applied to high-performance simulations. An interesting use of reversible computing is in its application to the automatic differentiation of functions expressed in a high-level computer language, such as $\mathbf{C}/\mathbf{C}^{++}$[Griewant et al. 1996; Grimm et al. 1996]. For this, reverse execution of certain intermediate computations is necessary, which is achieved via operator-overloading techniques of $\mathbf{C}^{++}$.

The state-saving techniques presented in [Gomes 1996] 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 for simulation, but no performance results are provided. Our work starts where [Gomes 1996] ends, and is concerned with techniques for minimizing the state size for realizing reversibility, and simultaneously minimizing the runtime execution overheads. Finally, in [Umamageswaran et al. 1998], 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, without adequate care, it can degenerate to traditional state-saving if a 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.

There is a commonly implemented optimization in copy state-saving: 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. Advanced inter-event analysis is necessary in order to reduce such overhead in reverse computation.

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 reverse computation is natural. In these applications, automatic techniques are easily found that 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. A few of them are discussed next.

Open Issues

In general, reverse computation reduces the overhead in the forward computation path, but potentially 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.

Acknowledgements

The authors would like to thank P. L'Ecuyer for his insights on the reversibility of random number generators, 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 Rajive Bagrodia for his technical comments on making a performance comparison between reverse computation and periodic state-saving.

REFERENCES

Bennett, C. 1982. Thermodynamics of computation. *International Journal of Physics*, *21*, 905–940.

Biswas, B. and Mall, R. 1999. Reverse execution of programs. *ACM SIGPLAN Notices*, *34*, *4* (April), 61–69.

Bishop, P. 1997. Using reversible computing to achieve fail-safety. In *Proceedings of the* 8^{th} *Internal Symposium on Software Reliability Engineering (ISSRE 97)*, 182–191.

Bellenot, S. 1992. State skipping performance with the Time Warp Operating System. In *Proceedings of the* 6^{th} *Workshop on Parallel and Distributed Simulation (PADS '92)*, 53–64.

Baker, H. G. 1992. 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, 507–524.

Carothers, C. D., Fujimoto, R. M. and Lin, Y-B. 1995. A case study in simulating PCS networks using Time Warp. In *Proceedings of the* 9^{th} *Workshop on Parallel and Distributed Simulation (PADS '95)*, 87–94.

Carothers, C. D., Perumalla, K. S. and Fujimoto, R. M. 1999. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *Proceedings of the 1999 Winter Simulation Conference*, to appear.

Chen, W. and Udding, J. 1990. Program inversion: more than fun. *Science of Computer Programming*, *15*, *1* (January), 1–13.

Das, D., Fujimoto, R. M., Panesar, K., Allison, D. and Hybinette, M. 1994. GTW: a Time Warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, 1332–1339.

EPPSTEIN, D. 1985. A heuristic approach to program inversion. In *Proceedings of the* 9^{th} *International Joint Conference on Artificial Intelligence*, 219–221.

FRANK, M. 1999. The R programming language and compiler. `http://www.ai.mit.edu/~mpf/rc/home.html`

FUJIMOTO, R. M. 1989. Time Warp on a shared memory multiprocessor. In *Proceedings of the 1989 International Conference on Parallel Processing (ICPP '89)*, 242–249.

FUJIMOTO, R. M. 1990. Parallel discrete event simulation. Communications of the ACM, *33*, *10* (October), pages 30–53.

GOMES, F. 1996. Optimizing incremental state-saving and restoration. Ph.D. thesis, Dept. of Computer Science, University of Calgary.

GRIEWANT, A., JUEDOS, D., MITEV, H., UTKE, J., VOGEL O. AND WALTHER, A. 1996. ADOL-C: A package for the automatic differentiation of algorithms written in **C**/$\mathbf{C}^{++}$. *ACM Transactions on Mathematical Software*, *22*, *2* (February), 131–167.

GRIMM, J, POTTIER, L. AND ROSTIANG-SCHMIDT, N. 1996. Optimal time and minimum space-time product for reversing a certain class of programs. *Research Report, Institut National de Recherche en Informatique et en Automatique (INRIA)*,

LEEMAN, G. 1986. A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages and Systems*, *8*, *1* (January), 50–87.

L'ECUYER, P. AND ANDRES, T. H. 1997. A random number generator based on the combination of four LCGs. *Mathematics and Computers in Simulation*, *44*, 99–107.

MATSUMOTO, M AND NISHIMURA, T. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *8*, *1* (January), 3–30.

PERUMALLA, K. S. COOPER, C. A. AND FUJIMOTO, R. M. 1996. An efficiency prediction method for ATM multiplexers. In *Proceedings of Proceedings of the International IFIP-IEEE Conference on Broadband Communications*, 477–488.

PERUMALLA, K. S. AND FUJIMOTO, R. M. 1999. Source code transformations for efficient reversibility. *Technical Report, GIT-CC-99-21*, College of Computing, Georgia Institute of Technology.

PRESS, B. AND MACINTYRE, I. 1992. On the trade-off between time and space in optimistic parallel discrete event simulation. In *Proceedings of the* 6^{th} *Workshop on Parallel and Distributed Simulation (PADS '92)*, 33–42.

POPLAWSKI, A. AND NICOL, D. M. 1998. Nops: A conservative parallel simulation engine for TeD. In *Proceedings of the* 12^{th} *Workshop on Parallel and Distributed Simulation (PADS '98)*, 180–187.

THE REVERSIBLE COMPUTING HOME PAGE AT MIT 1999. `http://www.ai.mit.edu/~cvieri/reversible.html`

SOSIC, R. 1994. History cache: hardware support for reverse execution. *Computer Architecture News*, *22*, *5*, 11–18.

STEINMAN, J. S. 1993. Incremental state-saving in SPEEDES using $\mathbf{C}^{++}$. In *Proceedings of the 1993 Winter Simulation Conference*, 687–696.

UMAMAGESWARAN, K., SUBRAMANI, K., WILSEY, P. A. AND ALEXANDER, P. 1998. Formal verification and empirical analysis of rollback relaxation. *The Elsevier Science Journal of Systems Architecture*, *44*, 473–495.

VANDEN EYNDEN, C. 1987. *Elementary Number Theory*, Random House, New York.

XIAO, Z., UNGER, B., SIMMONDS R. AND CLEARY, J. 1999. Scheduling critical channels in conservative parallel discrete event simulation. In *Proceedings of the* 13^{th} *Workshop on Parallel and Distributed Simulation (PADS '99)*, 20–28.