Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

AN ALGORITHM FOR FULLY-REVERSIBLE OPTIMISTIC PARALLEL SIMULATION

Michael D. Peters
Christopher D. Carothers

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A.

ABSTRACT

Typically, large-scale optimistic parallel simulations will spend 90% or more of the total execution time forward processing events and very little time executing rollbacks. In fact, it was recently shown that a large-scale TCP model consisting of over 1 million nodes will execute without generating *any* rollbacks (i.e., perfect optimistic execution is achieved). The major cost involved in forward execution is the preparation for a rollback in the form of state-saving. Using a technique called *reverse computation*, state-saving overheads can be greatly reduced. Here, the rollback operation is realized by executing previously processed events in reverse. However, events are retained until GVT sweeps past. In this paper, we define a new algorithm for realizing a continuum of reverse computation-based parallel simulation systems, which enables us to relax the computing of GVT and potentially further reduces the amount of memory required to execute an optimistic simulation.

1 INTRODUCTION

As originally defined by Jefferson (Jefferson 1985), an optimistic parallel simulation will allow logical processes (LPs) to detect, rollback and cancel incorrectly processed and or scheduled events, and then resume normal forward event processing. Operationally, large-scale optimistic parallel simulations will spend 90% of execution time going forward, and very little time executing rollbacks. In fact it was recently shown that a large-scale TCP model consisting of over 1 million nodes will execute without generating *any* rollbacks(Yaun et al. 2003).

Furthermore, most rollbacks tend to be short in distance. Applying the principle of executing the common case fast, several rollback algorithms, such as Incremental State-Saving (Gomes 1996) and Infrequent State-Saving (Line and Preiss 1991; Lin et al. 1993) and Lookback (Chen and Szymanski 2003) have reduced either the preparation required for a rollback or relaxed the conditions under which a rollback may occur, thus realizing faster execution.

Forward execution has two significant overheads associated with preparing for rollback. First, each logical process (LP) must store substantial amounts of data to prepare for a rollback, resulting in many RAM accesses. This memory must be reclaimed during execution otherwise large simulations would require inordinate amounts of memory. Because optimistic protocols only reclaim the storage that is no longer needed, Global Virtual Time (GVT) must be computed, which is a lower bound on all unprocessed events in the system. To maintain efficient execution (i.e., lower memory overheads), GVT is re-computed very often. GVT is determined by the timestamp of the "slowest" LP. An efficient algorithm, such as Mattern's (Mattern 1993), requires O(N) messages for N processors.

In the original Time Warp algorithm, every LP would store its entire state before executing each message. Other techniques, such as Incremental State Saving (Gomes 1996), Infrequent State Saving (Lin and Preiss 1991), Rollback Relaxation (Umamageswaran et al. 1998) and Lookback (Cheng and Szymanski 2003) are "state-based" approaches to reducing these costs.

In this article, we investigate an alternative computation-based technique called **reverse computation**. Here, models are able to execute both forward and backward in simulated time. We divide reverse computation models into two classes: **time-proportionate** and **perfect**. **Time-proportionate reverse computation (TiPRC)** (Frank 1999) stores only the minimum set of control information generated by the forward processing of an event. One can view this control information as the "entropy" produced by the processing of an event. In TiPRC simulations, reverse execution is limited by how much garbage is produced. Once memory is exhausted, the garbage must be reclaimed before forward execution can resume, eliminating the possibility of reversing to the beginning. This technique is used to support the "undo" operation in optimistic parallel simulation. TiPRC has been shown to reduce the state memory requirements of optimistic simulations by a factor of 100 and increased the overall speedup by a factor of 6 when compared to classic state-saving or logging techniques (Carothers, Perumalla and Fujimoto 1999).

Perfect reverse computation (PRC) is a radical extension of TiPRC that produces no effective "entropy" in the forward or reverse processing of an event, thus allowing the simulation to run arbitrarily long in a constant amount of space. Our hypothesis is that under PRC, events in the optimistic simulation can be immediately committed. Instead of "rolling events back," simulation objects shift from forward processing to reverse processing. The immediate advantage of this modeling methodology is that global virtual time (GVT) calculations and fossil collection (i.e., garbage collection) are no longer needed. Thus, PRC, under certain conditions, holds the promise of allowing large-scale network models to scale to much larger processor configurations than TiPRC-based parallel simulation systems.

The contribution of this paper is to define and properties for the PRC class as well as an algorithm for realizing a continuum of PRC to TiPRC-based parallel simulation systems. In the next section, we describe more fully these computation classes and their respective conditions.

2 REVERSE COMPUTATION CLASSES AND CONDITIONS

2.1 TIPRC

To illustrate the TiPRC approach, we begin with a simple example. Consider a simple model 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, Cooper and Fujimoto 1996).

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.

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.

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

bit b1;
```

(a) State

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

(b) Forward code

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

(c) Reverse code

Figure 1: ATM multiplexor model with TiPRC

Now, to support TiPRC, we have also added a single **bit** variable to the state of the multiplexor. This variable is used to note whether the `if` statement was executed or not (i.e., `b1` $= 1$ if `qlen` $<$ `B` and 0 otherwise).

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

2.2 TIPRC PROPERTIES

We can make some observations to understand some of the properties of of TiPRC models.

- **Property 1:** The majority of the operations that modify the state variables are "constructive" in nature. That is, the undo operation 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. The $*=$ and $/=$ operators require special treatment in the case of multiply or divide by zero, and overflow/underflow conditions. More complex operations such as *circular shift* (*swap* being a special case), and random number generation also belong here.

- **Property 2:** The complexity of the code is such that the "control state" occupies less memory than the "data state" of the variables. In cases where the code is equally or more complex than the data state, reverse computation will not pay-off relative to the previous mentioned state-based approaches.
- **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 4:** Regardless of the reversibility of LP state, messages *must* be retained in order to support reverse event processing.

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 degenerates to 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.

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 have shown that TiPRC is well suited for the optimistic simulation of large-scale network models. We believe the same will be true of PRC.

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. As example, L'Ecuyer's Combined Linear Congruential RNG (L'Ecuyer and Andres 1997) is perfectly reversible without resorting to any state-saving despite making use of individual operations such as integer division that result in bit loss (Carothers, Perumalla and Fujimoto 1999)

Property 4 provides the limitation on how far a parallel simulation can be reversed under TiPRC. In particular, a complete set of events must be retained in order to undo event processing. Thus, as time moves forward, the amount of space that *must* be retrained grows. Because space in practice cannot grow without bounds, the messages are what makes this class of reverse computation only time-proportionate. So then, is it possible to relax that requirement and ultimately support PRC? As we will show in the next section, it is possible if certain properties within the model hold.

2.3 PERFECTLY REVERSIBLE COMPUTATION

As previously discussed, PRC is an extension of TiPRC which generates no entropy in the form of information loss. Recall, that TiPRC generates entropy in the form of control information (i.e., records which branch was taken or how many times through a loop plus incrementally state-saves any destructive assignments). This control information was stored in a bit field. PRC has no such bit field. PRC is able to execute forward or backward based solely on the current state of the system.

To support PRC in general, a program cannot produce any information loss. This means PRC should allow arbitrarily-long reversible computations through simulated time consuming *only a constant amount of memory*. This is the fundamental difference between PRC and TiPRC. TiPRC creates garbage in the form of control bits and message data, thus limiting reverse execution over a range of simulated time. Once the garbage bits are collected, TiPRC cannot reverse beyond that point in the computation. This constant amount of memory requirement leads to two additional properties that must hold in practice for PRC to be supported:

- **Property 5:** a set of independent "source" LPs must be defined which generate or push messages into the network of LPs. These messages are generated based on some stochastic distribution. This set of source LPs must never receive events from other LPs. They strictly schedule events to other LPs.
- **Property 6:** once a message leaves a source LP, the path it takes to the final destination or consumer LP should not contain a cycle.

Property 5 is required because this set of LPs is capable of fully producing the input state use by all other LPs. That is to say, should an LP need to rollback further than its history window allows, it can request past messages be re-sent for this source set of LPs. Because those LPs are independent, they are completely reversible because the random number generator is perfectly reversible as described in property 3.

Property 6 suggests that the path a message takes during its processing is well defined within the model and does not result in LPs being re-visited. We can relax this constraint by providing a full-storage LP (i.e., stores all messages and state) within the cycle. While this relaxation shifts the system back to a TiPRC execution model, we believe that in practice this is a small memory overhead.

3 TiPRC/PRC ALGORITHM

In this section, we describe the details of our TiPRC/PRC algorithm. The algorithm is broken down into series of functions which are shown in Figures 2 – 8. We begin with our terms and definitions.

3.1 TERMS AND DEFINITIONS

We introduce the concept of reverse-processing a message. Suppose processing a message M takes the LP from state S to state S', producing messages $P_1...P_n$. Reverse-processing message M will take the LP from state S' to state S, producing messages $P_n...P_1$. It is only allowable to reverse-process message M from state S', any other state would result in an incorrect execution.

When in a large rollback (also called a 'reversal'), an LP will pass around large amounts of messages, in the form of multi-messages. A multi-message is a linked list of messages, all with the same source and destination LP, and organized by decreasing timestamp, with the largest timestamp at the head of the list. These properties allow us to take a set of multi-messages and create a total list for a destination LP in timestamp order by performing a merge which consumes $O(N)$ time to sort N messages, assuming relatively few chains compared to the number of messages.

The maximum offset of a message is the largest possible difference between the timestamp of an incoming message and the timestamp of the outgoing message it generates. For normal (Gaussian) distributions, this is twice the mean. For other distributions, this may be infinity a with very small probability. For these cases, treat the maximum offset as the largest offset with reasonable probability. The maximum offset of an LP is the largest maximum offset of all the message types it handles.

The time of a rollback is the "rollback window". So, if an LP is at time T1 and has to rollback to time T0, the rollback window is {T1 to T0}. When we perform a large rollback, we require that some LPs have stored, or can recreate, messages back to GVT. These are called 'full storage LPs', and the number of LPs between them is the frequency of full storage LPs.

```
// inserts message M into an already sorted list,
// B. MAX_DECREASE is the maximum number of
// messages deleted at any one time.
enqueue( message_buffer B, message M)
   {
   insert M into B, maintaining sorted order;
   for( i =  0; i < MAX_DECREASE; i++)
      {
      if( (timestamp of M) -
          (timestamp of 2nd smallest message of B)
           <= lookback)
         pop smallest message from B;
      else
         break;
      }
   }
```

Figure 2: Lookback Algorithm

In reversing, we must send instructions between LPs and Processor Elements (PEs). Instructions are similar to messages, except they are stored in a separate place, and read separately in the scheduler (see Figure 3).

3.2 LOOKBACK

Each LP stores messages locally to execute a small rollback. The window of this storage is called the 'lookback'. Please note, this formulation of lookback similar to the Lookback protocol described in (Chen and Szymanski 2003). There, lookback is used as a relaxation condition for determining when an LP should rollback. Here, it is used as a storage window which is defined as a certain number of messages, or it can be all messages over a certain amount of simulation time. The latter is preferable, because rollback likelihood is related to the simulation time and not the number of messages.

The actual size of the lookback will vary depending on the physical specifications of the system, but it must be larger than the maximum offset of a message, otherwise one message can clear the memory. If the simulation uses a distribution with an infinite maximum offset, then receiving a message at the maximum offset will clear the queue, making a large rollback very likely. Similarly, if a simulation uses a distribution with a very large, but very rare, maximum offset, it may be desirable to make the lookback shorter than the maximum offset, resulting in the same problem. To prevent a single message clearing the queue, set the maximum number of messages discarded (MAX_DECREASE) relatively low. If the message at the maximum offset is not rolled back, each of the subsequent messages will also free multiple messages, and soon the message buffer will be the appropriate size. Other degenerations arise when a message buffer is very small (fewer than 3 messages). By imposing a very small minimum number of messages, we can avoid these.

Observe here, we do not require GVT to free memory. Instead, we can just shorten the lookback on some LPs to free memory. Good candidates would be the LPs that are farthest back, because they are unlikely to roll back, or those LPs that have an LP with full storage as a source, so their full rollback will be the least expensive. Occasionally, we should calculate GVT to free messages from the LPs with full storage, but because we are storing fewer total messages, this calculation will be less frequent.

3.3 LARGE ROLLBACK

We assume message latency to be low, and the time to process any single message to also be low, which is typical of network models. We also assume that the layout of the simulation is known from the start, that is, we know the possible sources of every LP, and they do not change during the simulation.

A rollback begins in the scheduler (see Figure 3), when an LP R receives a message with timestamp T smaller than the current time of R. This calls `start_rollback(R, T)` shown in Figure 4. If R has messages stored locally back to T, it will rollback normally, as in ROSS (Carothers, Bauer and Pearce, 2002). Otherwise, it initiates a large rollback, as shown in Figure 7.

As shown by the algorithm in Figure 7, A large rollback occurs when local storage is insufficient to handle a rollback. It requires that we recreate the messages back to T. For describing this algorithm, assume a simple network model, with sources, layers of routers in sequence and sinks, with no loops. The routers are organized in layers, so there is no router which is both upstream and downstream of any other router. This is a simplified model, but it easily extends to more complex systems. In a loop in the message sequence, we require that one LP be full storage in the loop.

R has already checked that it cannot handle the rollback locally. It now sends out instructions to resend messages back to time T. These instructions go to each of its sources that have sent it a message within the rollback window. The scheduler loop catches the instructions, and calls `resend_messages(S,T)` shown in Figure 5 on each source S.

If those sources S have full storage, they will be able to fulfill the request. In this case, they call an application-specific function to recreate the messages, or recall them from storage, and forward them as a multi-message. The multi-message contains every message from S to R back to time T, along with the message immediately before time T, as a confirmation that there is nothing missing from the chain. Reaching a full storage LP is the termination condition for requesting resends, and every chain of sources must terminate in a full storage LP.

Any of those sources S that do not have full storage will request messages from their sources. In order to ensure correct ordering of messages, we must process extra messages. The window of the extra messages must equal the maximum offset of the LP, so the request must be to time $T - M$, where M is the maximum offset. This is handled in the same way as the initial request from R. This repeats, with an additional offset at each repetition, until every request reaches an LP with full

```
scheduler()
   {
   retrieve, sort and execute PE instructions,
      ordered as follows;
   switch( current PE instruction type)
      {
      case Reversal_Starting:
         increment counter
            for number of LPs reversing;
      case Reversal_Finishing:
         decrement the number of LPs
            on this PE that are reversing;
         if( no LPs are still reversing )
            for( each LP on this PE)
                finish_reversal( LP );
      }
   retrieve, sort and execute reversal instructions;
   resend_messages(L, T);
   retrieve resent messages, and
      insert in destination LP's queue;
   if( Destination LP L received a resend
       for each request it sent)
      {
      if( L is the originator of the large rollback)
         execute_large_rollback(L);
      else
         rev_process(L);
      }
   if( any LP on this PE is still reversing)
      go to top of scheduler;
   if( another large reversal is pending)
     execute it;
   retrieve all other normal pending event messages;
   check each message for cancellation
   execute messages, checking for rollbacks;
   if( rollback occurs)
      start_rollback(LP, time);
   }
```

Figure 3: Modified Optimistic Simulation Engine Event Scheduler

```
// invoked on a primary or 2ndary rollback
start_rollback(lp L, timestamp T)
   {
   if( L has storage to rollback before T}
      rollback locally;
   else
      {
      send PE message ‘‘Reveral_Starting’’;
      store the state of L;
      for each (source S of L)
         if(S sent L a message with
            timestamp >= T )
            send a reversal instruction to S;
      }
   }
```

Figure 4: Start Rollback Routine

```
resend_messages(lp L, timestamp T)
   {
   // called when L receives a
   // ‘‘resend’’ instruction
   if( L is full storage)
      {
      if(has storage to request time)
         send stored message upto
            request time;
      else
         call app-specific function to
         recreate messages;
      }
   else
      {
      for each (source S of L)
         if((S sent L a message with timestamp) >=
            (T - maximum offset of L))
            send a reversal instruction to S;
      }
   }
```

Figure 5: Resend Messages Routine

storage.

When an LP L other than R has received as many resend multi-messages as it sent requests, the scheduler calls `rev_process(L)` shown in Figure 6. L reverse-processes each message, in order of decreasing timestamp. The output messages are stored in a windowed queue, sorted by timestamp. We use a windowed queue because most messages have a random offset, that is, they are scheduled at a random time in the future, and we need to ensure correct ordering of output messages. For example, assume we have two messages, one at time 10 and one at time 9. We would first reverse-process the event at time 10, which might get an offset of 2, scheduling an event at time 12. Then we would reverse-process the event at time 9, which might get a larger offset of 5, scheduling an event at time 14. If we naively sent each message along, the message at 12 could be processed before the message at 14. Furthermore, if we did not get around to processing the message at 9, the message at 14 would be dropped entirely!

```
rev_process( lp L )
   {
   // invoked when L recieves all the resend messages.
   // LP L has multi-messages from it source LPs
   // stored in a local heap sorted by timestamp.
   remove messages with timestamp >=
      current time of L;
   for each( message M sent to L )
      {
      call app-specific function to
         reverse-process M;
      push returned messages onto windowed queue
         to ensure correct ordering;
      as the window moves, popped messages
         are placed in multi-messages sorted by
         destination LP;
      }
    //window is equal to the maximum offset of L
    if( the multi-message from that source
        is exhausted )
       {
       record the exhausted source, return;
       }

    send the multi-message chains created
       from the windowed queue;
    }
```

Figure 6: Reverse process routine.

To avoid this problematic behavior, the queue has a window the size of the maximum offset. Messages are processed in decreasing timestamp order, so message i has timestamp larger than message $i+1$. At worst, message i could get the maximum offset, and message $i+1$ could get no offset. But the difference between the two output messages is still less than the maximum offset, and the window on the buffer is the size of the maximum offset, guaranteeing ordering (remember, the queue is also sorted by timestamp). When a message is inserted in the queue, it moves the window up, pushing other messages off the end.

As each message comes off the queue, they are sorted by destination LP, giving us a set of chains, ordered by decreasing timestamp and sorted by destination LP. When the messages from one source are exhausted, we send all of the processed multi-messages. We cannot

```
execute_large_rollback(lp L)
   {
   // invoked when LP L is reversing
   // and has all ‘‘resend’’ messages.
   // L has multi-messages from it source LPs
   // in a local heap sorted by timestamp.
   remove messages with timestamp >= current time of L;
   for each( message M sent to L )
      {
      call application-specific function
         to reverse-process M;
      send returned messages as anti-messages;
      if( the multi-message from that source
          is exhausted )
         {
         if( L has rolled back far enough )
            {
            for each PE
               send PE instruction of
               type ‘‘Reversal_Finishing’’;
            }
      else
         send a reversal instruction of type
         ‘‘Resend_More’’ to the exhausted source;
      }
   }
```

Figure 7: Routine for Processing Large or Long Rollbacks

```
finish_reversal(lp L)
   {
   delete extra messages generated in rollback;
   if(L has stored state)
      restore L to the saved state;
   }
```

Figure 8: Routine for Completing a Reversal Rollback

continue processing when one incoming chain is exhausted because we can no longer guarantee the correctness of ordering of the messages. The exhausted source is recorded, in case the processed messages are insufficient and we need to request more messages to reverse-process farther back.

When R has received as many resend multi-messages as it sent requests, the scheduler will call `execute_large_rollback(R)` (see Figure 7. The rollback is executed just like `rev_process`, except that rather than producing normal messages to resend, it produces anti-messages. If the messages run out before R has rolled back far enough, it requests more messages from whichever source's chain was exhausted. Secondary rollbacks are handled in the same way as the primary rollback: if they can be handled locally, they are, otherwise we request resends.

Once the rollback is completed, R sends out PE instructions indicating that the rollback is finished. The scheduler calls `finish_reversal` on every LP on the PE. Those LPs that assisted in the rollback by recreating messages, but did not rollback themselves, restore their state to the state that was saved immediately before the rollback, and all LPs free resent messages. If another rollback is waiting for execution, it is handled now, otherwise we resume forward execution.

3.4 COMPLEXITY

Assume that processing a message takes $O(1)$. For a local rollback of N messages, it takes $O(N)$ time. Also assume the application-specific function to reproduce or recall the messages takes $O(N)$ for N messages.

Resent messages can be organized in trees and priority queues, so all operations are $O(log(N))$ for N messages. Sorting is done in two places, when the message is first produced, and when we combine the resend multi-messages from all the sources. The sorting when the message is produced is limited to the window of the buffer, which is the size of the maximum offset. This is small compared to the total number of messages we will process, so we can treat an insertion as $O(1)$, meaning that processing N messages is $O(N)$. Combining the multi-messages is done as the last stage of merge sort: given a set of chains, we remove the smallest item from the top, process that, and repeat. The number of chains is equal to the number of sources, which is much less than the total number of messages being processed, so again, we can treat this whole operation as $O(N)$.

The number of messages is related to how many layers are between the present LP L and the reversing LP R. R will have to process N messages, as in a local rollback. The first layer back, assuming it is not full storage, will have to handle $R+M$ messages, where M is the average number of messages in the span of the maximum offset. At each layer, we add another M messages to each LP. Additionally, we have the number of sources per LP to consider: if R has S sources, then each of them will have to process $R+M$ messages, giving us a total workload for the layer of $S(R+M)$. Fortunately, if an LP receives resend requests from multiple LPs, it can handle all the requests simultaneously, so the S term does not multiply beyond the size of a layer of sources.

The total complexity is $O(SLR+\sum_{l=1}^{L} SM*l)$, where S is the size of a layer of sources, L is the total number of layers of the rollback, and M and R are as defined above.

4 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; Frank 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 (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 toward parallel simulations, they are not optimized for minimizing the state size, and do not adequately exploit the semantics of constructive operations.

Reversible computing has also been suggested as a method for testing failures in real-time systems (Bishop 1997). 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 `C/CPP`(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 `CPP`.

Finally, in the context of optimistic parallel simulation, time-proportionate reverse computation has been applied to circuit models (Perumalla and Fujimoto 2002), and large-scale Internet models (Yaun et al. 2003).

5 CONCLUSIONS AND FUTURE WORK

We present an algorithm to allow both TiPRC and PRC-based optimistic parallel simulation. Using this, we can reduce memory overheads, and GVT calculations. The result ideally will be faster forward execution at the cost of the very rare large rollback. With the basic theory complete, we are implementing the algorithm in a real optimistic simulation system. One of the challenges of this implementation is ensure we get the "end cases" correct with respect to all the lists as well as ensuring the properties of the test model hold true. At this time, validation of model properties can only be done via visual inspection. We are currently investigating ways to automate the reverse code generation and property validation of TiPRC and PRC classes. The target applications for this algorithm are large-scale network models where the primary focus is the core network which allows traffic sources to remain relatively independent.

REFERENCES

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.

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

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

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

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Computer Modeling and Simulation*, 9 (3): 224–253.

Carothers, C. D., D Bauer and S. Pearce. 2002. ROSS: A High-Performance, Low Memory, Modular Time Warp System, *Journal of Parallel and Distributed Computing*, #62: 1648–1669.

G. Chen and B. K. Szymanski. 2002. Lookback: A New Way of Exploiting Parallelism in Discrete Event Simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, 153-162.

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

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

Michael Frank. 1999. Reversibility for Efficient Computing. Ph.D. thesis, MIT/LCS. Available online at http://www.www.cise.ufl.edu/~mpf/manuscript

F. Gomes. 1996. "Optimizing Incremental State-Saving and Restoration." Ph.D. thesis, Dept. of Computer Science, University of Calgary.

A. Griewant, D. Juedos, H. Mitev, J. Utke, O. Vogel and A. Walther. 1996. ADOL-C: A package for the automatic differentiation of algorithms written in C/CPP. *ACM Transactions on Mathematical Software*, 22(2): 131–167.

J. Grimm, L. Pottier and N. Rostiang-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. *Research Report, Institute National de Recherche en Informatique et en Automatique (INRIA).*

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

Y-B. Lin and B. R. Preiss. 1991. "Optimal Memory Management for Time Warp Parallel Simulation", *ACM Transactions on Modeling and Computer Simulation*, 1(4): 283–307.

Y-B. Lin, B. R. Press, W. M. Loucks, and E. D. Lazowska. 1993. "Selecting the Checkpoint Interval in Time Warp Simulation". *In Proceedings of the* 7^{th} *Workshop on Parallel and Distributed Simulation (PADS '92)*, 3–10.

D. R. Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3): 404–425.

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

F. Mattern. 1993. Efficient distributed snapshots and global virtual time algorithms for non-FIFO systems. *Journal of Parallel and Distributed Computing*, 18(4): 423–434.

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

K. Perumalla, and R. Fujimoto. 2002. Using Reverse Circuit Execution for Efficient Parallel Simulation of Logic Circuits, In *Proceedings of The International Society for Optical Engineering (SPIE) Annual Meeting.*

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

K. Umamageswaran, K. Subramani, P. A. Wilsey and P. Alexander. 1998. Formal verification and empirical analysis of rollback relaxation. *The Elsevier Science Journal of Systems Architecture*, 44: 473–495.

G. Yaun, C. D. Carothers and S. Kalyanaraman. Large-scale TCP models using optimistic simulation, In *Proceedings of the* 17^{th} *Workshop on Parallel and Distributed Simulation*, June 2003.

ACKNOWLEDGMENTS

This research is supported by an NSF CAREER Award CCR-0133488, and the DARPA Network Modeling and Simulation program.

AUTHOR BIOGRAPHIES

MICHAEL D. PETERS is a PhD student in the Computer Science Department at Rensselaer Polytechnic Institute. He recieved his BS from Rensselaer Polytechnic Institute in 2001. In summer 2003 he interned at Sandia National Labs, developing a distributed genetic programming engine. In summer 2002, he interned for GE/CRD developing biocomputation simulations. His research interests include parallel and distributed systems, simulation systems, and quantum computation. His email address is <peterm3@cs.rpi.edu>.

CHRISTOPHER D. CAROTHERS is an assistant professor in the Computer Science Department at Rensselaer Polytechnic Institute. He received the PhD, MS, and BS from Georgia Institute of Technology in 1997, 1996, and 1991, respectively. Prior to joining RPI, he was a research scientist at the Georgia Institute of Technology. As a PhD student, he interned twice with Bellcore, where he worked on wireless network models. In 1996, he interned at MITRE Corporation, where he was part of the DoD High Level Architecture development team. His research interests include parallel and distributed systems, simulation, and networking. His email address is <chrisc@cs.rpi.edu>.