An Approach for Incorporating Rollback Through Perfectly Reversible Computation in a Stream Simulator

David W. Bauer and Ernest H. Page
The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102, U.S.A.
{dwbauer,epage}@mitre.org

Abstract

The traditional rollback mechanism deployed in optimistic simulation is state-saving. More recently, the method of reverse computation has been proposed to reduce the amount of memory consumed by state-saving. This method computes the reverse code for the model during rollback execution, rather than recalling saved state memory. In practice, this method has been shown to offer memory-efficiency without sacrificing computational efficiency. In order to support reverse codes in the model, events must continue to be preserved in the system until fossil collection can be performed. In this paper we define a new algorithm to support perfectly reversible model computation that does not depend on storing the full processed event history. This approach improves memory consumption, further supporting large-scale simulation.

1 Introduction

Jefferson describes an algorithm for parallel discrete event simulation (PDES) that allows for optimistic execution of events [16]. In order to preserve the causality constraint (see [17]), optimistic execution requires a mechanism for *rolling back*, i.e., undoing the effects of out-of-order event processing. Typically, rollback mechanisms are based on *state-saving*. These approaches include Lazy Cancelation [28], Incremental State Saving [13], Infrequent State Saving [21, 22], Fast-Software Checkpointing [25], and more recently Lookback [8] and Batch Cancelation [31]. In each approach, Logical Process (LP) state is saved prior to the processing of an event. When state sizes are large, or frequently modified, the cost of state saving can quickly become a barrier to large-scale optimistic simulation.

Carothers, Perumalla and Fujimoto [7] suggest Reverse Computation (RC) as an alternative approach to rollback. In RC, events are saved but LP states are not. Compiler techniques are used to automatically generate "reverse code" for events which is executed to reconstruct LP states during rollback. This approach has been implemented in simulators such as GTW [9], ROSS [3] and μsik [24], and has been applied to large-scale, packet-level network simulations [4, 29, 30] and circuit design models [23]. Most recently RC has shown excellent performance results within the particle-in-cell method for physics simulation [26].

Li and Tropper [20] observe that in some cases, e.g. VLSI simulations, event sizes (rather than state sizes) can be very large, and therefore event saving overheads can have a significant impact on model execution. They suggest a method for event reconstruction using saved LP states in a Cluster Time Warp implementation [1].

So what is the intrepid parallel simulation engine designer to do? Save states? Save event message data? Wouldn't it be nice if you didn't have to save either – at least for some models?

In this paper, we suggest an approach to *perfectly reversible computation* (PRC) which supports rollback in an optimistic simulation using only the current LP state and current event. The approach is defined in the context of the Adirondack stream simulator developed by Bauer and Carothers [2].

The remainder of the paper is organized as follows. Section 2 provides brief review of reverse computation, and our suggested definition of the concept of perfectly reversible computation. Some relevant details for stream simulation are given in Section 3. Section 4 outlines the implementation of PRC in a stream simulator. Section 5 presents an example in the context of a well known network model. Conclusions and areas for future work are given in Section 6.

2 Reverse Computation

Because it offers a possible solution to the heat problem faced by chip manufacturers, reversible computing (RC) has been extensively studied in the computer architecture arena. The promise of reverse computation is that the amount of heat loss for reversible architectures would be minimal for significantly large numbers of transistors[18, 27]. Rather than creating entropy (and thus heat) through destructive operations,

a reversible architecture conserves the energy by performing other operations that preserve the system state [5, 10].

In the software realm, reverse computation concepts have been successfully applied in areas such as database design [12], checkpointing and debugging [6] and code differentiation [14, 15].

2.1 Reverse Computation for PDES

Based on the successful application of RC concepts in other software domains, Carothers, Perumalla and Fujimoto [7] suggest the application of RC to reduce state saving overheads in PDES. They define an approach based on reverse event codes (which can be automatically generated), and demonstrate performance advantages of this approach over traditional state saving for fine-grained applications (those with a small amount of computation per event).

As noted in [7], the key property that reverse computation exploits is that a 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. Note, that 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 certain classes of random number generation also belong here.

Operations of the form $a = b$, modulo and bit-wise computations that result in the loss of data, are termed to be *destructive*. Typically these operations can only be restored using conventional state-saving techniques. However, we observe that many of these destructive operations are a consequence of the arrival of data contained within the event being processed. For example, in [29], the last-sent time records the time stamp of the last packet forwarded on a router LP. We use the swap operation to make this operation reversible.

2.2 Perfectly Reversible Computation

In typical applications of RC to support rollback, events must be stored to support state reconstruction. As Li and Tropper [20] observe, event storage can represent a significant overhead. We propose the term *perfectly reversible computation* (PRC), to indicate models that do not require the storing of previously processed events. In addition to restoring LP state, PRC models must reconstruct the previously processed events in the LP event history.

To support PRC, only two elements are used to reconstruct the previous LP state: the current event being rolled back, and the current LP state. In PRC, event data are no longer stored, but instead immediately reclaimed by the system.

This definition of perfectly reversible computation indicates that certain techniques for reverse computation, as applied in [7], can no longer be supported, namely, (1) using a per-event bitfield to indicate the control state traversed during forward execution of an event; and (2) using the swap operation to save state values in the event message.

A per-event bitfield is used in [7] to determine the branching that occurred during the forward processing of an event. Branching includes statements such as: if..else, goto, switch, function calls and loop constructs. For example, during the forward processing of an event, a bit in the bitfield of the event would be given a value of 1 to indicate that the conditional in an if..else statement resolved to TRUE, and given the value 0 otherwise. During rollback, the correct path through the code to reverse compute could be determined by examining the event bitfield.

The second RC technique unavailable to PRC is the swapping of state variables with event message variables. The swapping technique was previously used to overcome the irreversibility of *destructive assignments* in general (non-PRC) models. Examples of destructive assignments include statements such as plain assignments and floating point operations where precision information is lost.

Fortunately, a majority of the operations in a model are constructive in nature (i.e., ++, –, etc). Also, models typically exhibit highly predictable behaviors, and the previous state can be restored from the current event and LP state through a high level examination of the code. Further, some destructive statements can be converted to constructive statements. For example, a level of precision can be defined for floating point operations (in terms of decimal places) and then all occurrences shifted into into the integer domain (e.g., we define a precision to 6 decimal places and represent 15 milliseconds is represented as the unsigned integer value 1500 and not the floating point value 0.0015, yielding a range of 0.0 to 4,294 seconds). In Section 5 we illustrate an example of this type of model.

3 Stream Simulation

PDES implementations traditionally consist of a collection of *logical processes* or LPs, where each LP represents a distinct component of the system being modeled. LPs communicate by exchanging timestamped messages (events). The LPs are assigned to processing elements (PEs), where each PE represents a single computational resource in hardware, as shown in Figure 1.

The mapping of LPs to PEs has an impact on the performance of the simulation. If the frequency of messages exchanged between LPs mapped to different PEs is too high, then communications overhead can dominate the simulation performance characteristics. We use the term *remote event* to refer to events sent between LPs on separate PEs. In the synthetic workload benchmark model, PHOLD [11], the communication patterns between LPs are uniformly distributed across the LPs, and the LPs are uniformly mapped to PEs. Fixing the LP population within the model, as more PEs are added to the simulation, the frequency of events passed between PEs increases.

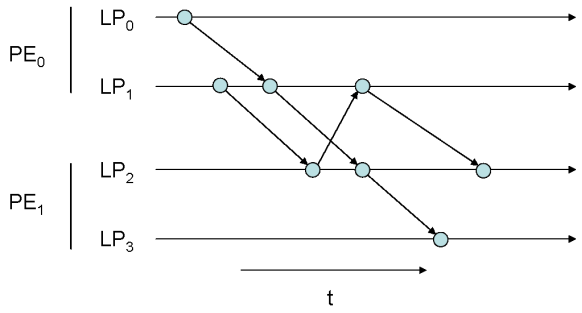

Figure 1. LP to PE mapping in traditional Time Warp simulator. LPs 0 and 1 are mapped to PE 0, and LPs 2 and 3 are mapped to PE 1. Events sent between LPs mapped to PE 0 and LPs mapped to PE 1 require synchronization.

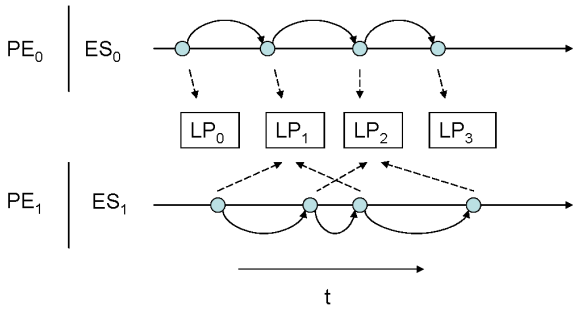

Figure 2. Event stream (ES) to PE mapping in stream simulation. Events are no longer sent between PEs. Instead, any PE may execute an event for any LP destination. Event stream 0 is defined as the initialization event created by LP 0, and each subsequent event created as a subsequence of processing that event.

Bauer and Carothers [2] suggest *stream simulation* as an implementation of PDES to minimize (actually eliminate) remote events. An *event stream* is a set of events, S, constructed by placing an initialization event, e_i, in the set, and computing the transitive closure of all events that are *scheduled by* events in S. That is, if an event $e_x \in S$ and event e_x schedules events e_y and e_z, then $e_y \in S$ and $e_z \in S$.

Rather than mapping LPs to PEs, a stream simulation maps event streams to PEs. Figure 2 illustrates how event streams (ES) are mapped to PEs in a stream simulation.

From the aspect of inter-process communication, the cost of synchronization in stream simulation can be far less than in the traditional LP-to-PE approach. However, while there are no remote events in a stream simulator, synchronization *is* required around LP states. Bauer and Carothers [2] show this synchronization cost to be statistically insignificant for models with large numbers of LPs, and a relatively small number of PEs (e.g., utilizing 8 PEs to execute a model containing 10,000 LPs results in a 0.003% probability that any two PEs will be processing events for the same LP simultaneously). So, in the spirit of optimizing for the typical case, stream simulation takes advantage of the very low probability that two PEs will attempt to process an event for the same LP simultaneously.

Computing the traditional PHOLD model on a quad-processor, dual-core Opteron server, and configured with both 10- and 20-million LPs, the Adirondack stream simulator was able to generate an almost 30% improvement in the runtime of the model over the ROSS simulator [2].

3.1 Rollback Mechanism

Because stream simulation allows any processor to commit events for any LP, processing events out of order at an LP may occur. Out of order event processing happens when event streams mapped to different PEs are "out of phase" in the LP timeline. For example, as illustrated in Figures 1 and 2, the first event in event stream 1 (which involves LP 1) should occur before the second event in event stream 0 (which also involves LP 1).

Traditional Time Warp simulators implement the rollback mechanism internal to a PE, and so only one PE is ever operating on the LPs mapped to the PE. In a stream simulation, LPs must be serialized during the rollback period. Further, to avoid "migration" of event streams across PEs, events canceled as part of a rollback are canceled by the PE that originally processed the event.

In a stream simulation, just as in a traditional Time Warp system, processed events can be stored in a per-LP queue to support future rollbacks if they occur. After processing an event, the PE stores the event in a per-LP processed event list in last in, first out (LIFO) order. In traditional Time Warp, there is no logical relationship between events in this list. But stream simulation creates a logical relationship between processed events. Each event in a stream is preceded by the event that created it (if each processed event creates only one new event). During the rollback of an event stream, the previous event in the stream could be reconstructed using the technique of reverse computation. Section 4 outlines the model and simulation executive modifications necessary to support event reconstruction.

3.2 A Note About Random Number Generator Seeds

In a traditional Time Warp system, random number generator (RNG) seeds are allocated on a per-LP basis. Based on the LP-to-PE mapping, an LP RNG is always executed by a single PE, and therefore is "parallel-safe". The operations invoked on an RNG must be safe from race conditions occurring in hardware in order for the simulation to execute deterministically.

Algorithm 1 LP rollback algorithm. Streams are marked as needing rollback if the stream variable `RB_TO` has a value greater than the straggler event timestamp. `RB_TO` is set to -1 when not in the rollback state. If the stream is mapped to the current PE, then the rollback is processed immediately. If the stream is not mapped to the current PE, then the stream `RB_TO` is set and the owner's PE is notified to roll back the stream. Processing the straggler is delayed until all necessary streams have been rolled back.

Steps to Rollback an LP

```
for each stream in LP processed stream queue
    with stream->ts > straggler->ts
{
    remove stream from queue;

    if stream->RB_TO > straggler->ts
    {
        stream->RB_TO = straggler->ts;
        if stream mapped to local PE
            process RB;
        else
            notify owning PE of stream RB;
            enqueue straggler in
                    PE pending RB queue;
    }
}
```

In stream simulation serialization occurs around event streams, rather than LPs. Consequently, RNG seeds must be associated with event streams in order to preserve the deterministic property of the simulation.

Associating RNG seeds with event streams facilities event reconstruction. During the creation of a new event, if either the event destination LP, or the event timestamp are randomly determined, then these randomly generated values can be reproduced internal to the event stream.

4 Incorporating PRC with Stream Simulation

In this section we describe the details of our PRC algorithm in the context of a stream simulator implementation. We outline the supporting data structures in the simulation executive as well as show how to formulate PRC models.

In order to eliminate saving per-LP processed event histories, the rollback mechanism must be modified. Previously, we maintained lists of processed events that could be presented in reverse order to the LP RC event handlers during rollback. Correct rollback requires knowledge of: (1) the streams that executed events for the LP being rolled back, and (2) the timestamps of those events. The LPs may retain this information as a tuple stored in a *processed event stream queue*. Events stored in the processed event queue are now replaced by a tuples of a constant size. The tuple data is then used in a rollback to notify the owning PE that a stream must be rolled back to the required time.

Algorithm 2 Event stream rollback algorithm. An event stream is rolled back by re-creating each previous event in the stream, reverse computing LP states along the way. Care must be taken to ensure that each LP is reversed in the same order in which events were forward processed. RB is the rollback function, and RC is the reverse computation function.

Steps to Rollback an Event Stream

```
while stream->ts > stream->RB_TO {
    event e = stream->event;

    tw_event_cancel(e);

    if stream == next stream in LP
                processed stream queue
    {
        // Secondary Rollback
        call Algorithm 1
        break;
    } else
    {
        LP lp = e->dest_lp;

        stream->event =
            lp->RC(lp->state, e->data, lp);

        // re-enqueue event into PE PQ
        tw_pq_enqueue(me->pq, e);
        lp->ts = lp->processed_stream_qh->ts;
    }
}
```

4.1 Algorithm for LP Rollback

Algorithm 1 illustrates an LP rollback. Rolling back an LP is handled in a similar way using processed stream queues, with one major exception. Because we do not immediately have the event data required to rollback the LP state, we must enqueue the straggler event and wait for the stream to be rolled back by the PE to which the stream was mapped. Within the scheduler loop, pending stream rollbacks scheduled by remote PEs should be processed prior to normal event processing.

4.2 Algorithm for Event Stream Rollback

Algorithm 2 illustrates an individual event stream rollback. A PE only maintains the last processed event in each stream mapped to the PE. Each event caused by this event is currently pending processing, and so can be canceled immediately by the call to `tw_event_cancel`. If events have been processed by the LP after the candidate stream event, a secondary rollback occurs. The destination LP must first be rolled back, and the stream is left unmodified. If the last event processed

by the LP is this candidate, then the LP may reverse compute its state. The result of the reverse computation is the re-creation of the event that preceded this event in the stream. This event becomes the new candidate for rollback and the process repeats until all events in the stream with timestamp greater than or equal to `RB_TO` are reverse computed.

At the conclusion of the rollback, the event stream contains the last processed event prior to `RB_TO`, and any events created by that event should be pending processing in the PE data structures.

Deadlock cannot occur in the algorithm because PEs never block waiting for rollbacks to occur. PEs are free to continue optimistically processing other event streams while waiting for rollbacks. If a rollback where to involve all streams in the system then all PEs would be actively rolling back LP states to the required time. Because deadlock did not occur in arriving at this state, and causal order is preserved in the reversal of event processing, deadlock cannot occur.

4.3 Perfectly Reversible Random Number Generator

Consider again the PHOLD benchmark model. To reconstruct the previous event in a stream, two things must be known: (1) the source LP of the previous event; and (2) the scheduled timestamp of the previous event. We know the source of the current event during a rollback, and this is the same as the destination of the previous event. We can reconstruct the source of previous event by restoring the RNG seed used to create the previous event. Note that this is LP independent, as the seed is associated with the event stream. The second part involves determining the timestamp at which the previous event was scheduled. In PHOLD, this too can be restored from the RNG seed.

If the steps for constructing new events in the forward processing phase of the PHOLD model are:

```
FORWARD COMPUTATION:

ts = rng_exponential(Generator, Rate);
dest_lp = rng_uniform(Generator, 0, nlp);
tw_event_new(src_lp, ts, dest_lp);
```

where `ts` is the timestamp offset of the new event, and `dest_lp` is the destination LP. The functions `rng_exponential` and `rng_uniform` compute the next value in the exponential and uniform probability distribution functions, respectively, using the RNG seed indicated by Generator. The current LP is denoted by the `src_lp` variable. Then the steps for reconstructing the previous event in the stream are:

```
REVERSE COMPUTATION:

// reverse compute seed for dest_lp
rng_reverse(Generator);
//reverse compute seed for ts
rng_reverse(Generator);

// reverse compute for reconstruction
rng_reverse(Generator);
rng_reverse(Generator);

ts = rng_exponential(Generator, Rate);
dest_lp = rng_uniform(Generator, 0, nlp);
tw_event_recreate(dest_lp, ts, src_lp);
```

where `ts` is the timestamp of the previous event, `src_lp` is the current LP (and the destination LP of the previous event, and `dest_lp` is the source LP of the previous event. The function `rng_reverse` reverse computes the previous value of the RNG seed. The parameters to the function `tw_event_recreate` are the reverse of the function `tw_event_new`. Recall the current LP (`src_lp`) is the destination of the *previous* event in the stream. Note that the RNG seed is reversed an additional time to reconstruct these values, and then is left in the correct state for processing to continue in either the forward or the reverse for this stream.

The example assumes that a random number generator seed can be rolled back without resorting to state-saving. As noted in [7], for example, L'Ecuyer's Combined Linear Congruential RNG [19], reverse computes seed values without resorting to state-saving despite making use of individual operations such as integer division that result in bit loss.

Summarizing [19], reverse computation of RNGs is difficult because RNGs depend heavily on destructive statements. Information loss occurs because the algorithm depends on the semantics of integer division and modulo operations, and so previous seeds cannot always be derived from only the current seed values.

The high level abstraction used to construct this RNG examines the mathematical recurrence that defines the seed series and using its inverse for the basis of the reverse computation. In fact, it was shown that the computational requirements were the same for both the forward and reverse codes. Even without memory constraints, this RNG would be preferable when we consider the growing gap between PE and memory speeds.

5 A PRC Model Example: TCP-Tahoe

For PRC to work, it must be the case that the inverse of a model's code can be made entirely dependent on that code – at least for some models. We now give a *simplified* example of how to convert the TCP-Tahoe model presented in [29] to a PRC model. We focus on some of the primary functions in the model, illustrating the conversion.

5.1 TCP Event Handler

The TCP model event handler implements a simple, 3-state finite state machine. The full model details, including validation, can be found in [29]. The LP types are either client

or server, where servers send fragments of data to the clients, and the clients respond with acknowledgements (ACK). If a client ACK is not received by the server within a specified period of time, the server retransmits the last unacknowledged data packet.

The valid event handler states are: client, server, receiver time out (RTO). A TCP LP is defined as a client or a server for the complete runtime of the model. The third state implements the RTO timer and is a self-scheduled TCP server event to indicate that the client ACK has not been received in the required time period.

5.2 TCP Client

TCP clients in this model only receive events of one type: data. There are only two possible outcomes for this event: either the event is received in order, or it is not. The client then responds to the server with an event of type: acknowledgement.

The RC version of the TCP client examines the current event's sequence number to determine if the packet has been received out of order. This is a simple check of the state sequence number, and the event message number:

```
RC Client:

if(in_event->seq_num == state->seq_num){
    // in order packet recv'd
    bf->c2 = 1;
    state->seq_num += state->mss;
    state->out_of_order[...] = 0;
} else {
    // out of order packet recv'd
    bf->c3 = 1;
    state->out_of_order[...] = 1;
}
```

The bitfield `bf` is used to denote if this branch is taken. The reverse computation code simply inspects the bitfield and reverses the LP state for the branch taken in the forward code. The `seq_num` fields are fixed-width integers. The index of the `out_of_order` array contain zero destructive assignments or floating point operations.

We can construct the perfectly reversible logic of the branch instruction at a higher level. A value of 0 in the `out_of_order` array indicates a packet was received in the expected order; a value of 1 indicates an out of order packet was received. The PRC version of this branch is identical with the exception of the bitfield assignments in the forward execution. The PRC version of this branch is then constructed:

```
PRC Client:

if(state->out_of_order[...] = 0){
    // in order packet previously recv'd
    state->seq_num -= state->mss;
} else {
    // out of order packet prev recv'd
    state->out_of_order[...] = 0;
}
```

The next step in the PRC version of the model is to recreate the previous event in the stream from the current model state and the current event. The two values needed are the timestamp of the previous event, and the id of the LP that sent the current event. The LP that sent this event is statically defined for the runtime of the model, based on the TCP traffic topology, and so is known.

The current event timestamp was determined using the link bandwidth and delay values, that are static throughout the runtime. It is important to note that TCP LPs are interconnected via a network of IP-enabled router LPs. A problem arises when queue delays are dealt with on the sender's side of the link. This is information that is not available to the current LP. The solution here is to implement queue delays on the receiver's side. This is acceptable as links in this model are point-to-point, bi-directional, and full duplex. Then a call can be made to recreate the previous event using the API function:

```
tw_event *tw_event_recreate(tw_lp * dst,
    double offset_ts, tw_lp * src);
```

The `src` (current LP), `bf` and `offset_ts` being known, the final step is to re-create the application information. TCP servers only send data in response to client ACKS (or the init event which is never reversed). So the type of event to create is also known from the model semantics. The only remaining field in the model message is the `seq_num` of the previously acknowledged event. Luckily, since the current LP sent the previous ACK, the previous sequence number acknowledged is also known. It is then a simple matter to fill in the application data in the `event`.

Note that the simulation engine allocates an event from the free queue for the event re-creation. The current event can be immediately reclaimed at the conclusion of the reverse computation event handler, and the number of outstanding processed events remains constant.

5.3 TCP Server

TCP servers in this model only receive events of two types: acknowledgement and RTO. There are only two possible outcomes for ACKs, either the ACK is received in order, or it is not. If the ACK is in order, then the server is free to send the next packet of data. Otherwise, the server must resend the unacknowledged data packet. While the TCP server has additional states, each can be reached by the same method proposed in the client. Because of the similarity in the main control block of server ACK processing, we focus now on the RTO event type, or receiver timeout timer.

The RTO timer is constantly rescheduled by the server LP during data transmission. The RTO is realized as an event

scheduled into the future at the time when, if no ACK is received for a packet, then the packet is resent. The timer is then backed off by a factor of 2. Since packets are not sent without first receiving an ACK, this mechanism effectively decreases the send rate of the server, assuming congestion has occurred in the network.

Receiving the RTO event implicitly indicates that an ACK has not been received for a packet, and the LP state is updated accordingly, all of which occurs *constructively* in the model. The difficulty in reversing RTO events stems from the need to determine the previous time at which the RTO should fire. In the normal version of the TCP model, the previous timestamp of the RTO event is swapped with a field in the event. This is no longer possible due to the fact that model data are no longer stored by the simulator. Instead, the previous timestamp must be computed using only the current LP state, and the current RTO event being rolled back.

Because the RTO timer is always set for the oldest unacknowledged sent packet, the previous setting for the RTO timer can be determined from the state variable indicating the last acknowledged packet sequence number. It is a simple matter then to determine what the correct, previous RTO timestamp should be using only current state variables.

All that is necessary is to know how many events were sent between the unacknowledged event and the current time. Since data are sent in order, and only upon receiving an ACK, this can be determined by the congestion and receive window LP state variables. Using these state variables, the number of packets sent on receipt of the previous ACK can be computed. Again, we can re-compute the send time for each packet, back to the unacknowledged packet using the static link bandwidth and delay, thus recreating the RTO timestamp.

5.4 TCP Efficiency

The memory costs of storing processed events in the TCP model is high because as the packet traverses the IP network, each event processed must be saved until GVT sweeps past. A single packet traversing 8 nodes in the network causes 10 events to be stored in the simulator (8 IP + 2 TCP LPs). For large-scale models with 1 million concurrent flows, and each packet traverses 10 LPs on average, generates 10 million events stored in processed event queues. For a 32-bit architecture, a TCP message size is 108 bytes, translating into 1.054 GB of RAM consumed. The implication is that the model size will be limited by the amount of available memory.

While the computational costs of the PRC version is higher in the reverse code, the frequency of rollbacks in the TCP model were shown to be only a few percent. Minimal reductions were achieved in the computational cost of the forward code (i.e., we are no longer setting bits).

Figure 3 illustrates the impact of the memory savings, resulting in a 14% decrease in memory consumption due to storing only the last event processed. In addition, a >70% reduction in the size the event data was realized by removing fields required for state-saving and accounting for the tuples.

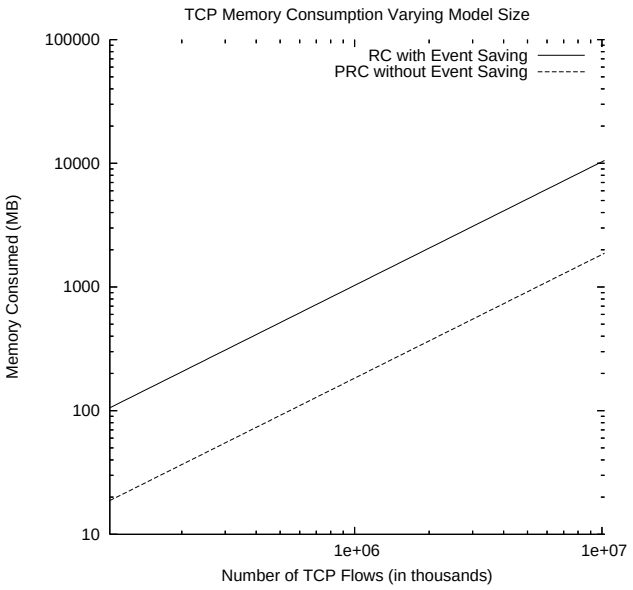

Figure 3. Event memory reduction using PRC-TCP model. Results shown are for a single packet traversing 8 IP routers for varying model sizes (number of simultaneously transmitting TCP server and client pairs).

For 1 million concurrent TCP Flows, the overall memory consumption for all streams is 187.5 MB of RAM, for a total reduction of 82.2%.

6 Conclusions and Future Work

We suggest an approach to incorporate rollback through perfectly reversible computation (PRC) in an optimistic parallel simulation based on event streams. We argue that using this approach, event memory overheads can be reduced to a constant factor per event stored. We outline model properties as well as simulation engine modifications required to support PRC and offer a simple example of PRC using a TCP-Tahoe model.

We are currently investigating porting other models to the PRC algorithm. We are particularly interested in large-scale models where the costs of storing event lists for rollback are prohibitively high, such as IP network models. The challenge is to identify and implement PRC procedures for model codes that are considered to be destructive, and replace them with codes that are able to re-create processed events. For example, the TCP model given here relied on a static TCP traffic topology, as well as a statically defined IP router topology. Supporting dynamic topologies is challenging because the model constructs no longer exist to support event reconstruction.

7 Acknowledgements

The authors would like to express their extreme gratitude to the Program Chair, the referees and shepherds for this article, who collectively went way beyond the call of duty to help the authors improve this paper. Their tremendous efforts are greatly appreciated.

The opinions expressed here are those of the authors and do not reflect official positions of The MITRE Corporation or the U.S. Department of Defense.

References

[1] H. Avril and C. Tropper. Scalable clustered time warp and logic simulation. 9(3):291–313, 1999.

[2] D. Bauer and C. D. Carothers. Eliminating remote message passing in optimistic simulation. In *WSC '06: Proceedings of the 38th conference on Winter simulation*. Winter Simulation Conference, December 2006.

[3] D. Bauer, S. Pearce, and C. Carothers. Ross: A high-performance, low memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 2002.

[4] D. Bauer, M. Yuksel, C. Carothers, and S. Kalyanaraman. A case study in understanding ospf and bgp interactions using efficient experiment design. In *PADS '06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 158–165, Washington, DC, USA, 2006. IEEE Computer Society.

[5] C. H. Bennett. The thermodynamics of computation. In *International Journal of Theoretical Physics*, volume 21 of *12*, pages 905–940, 1982.

[6] B. Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Not.*, 34(4):61–69, 1999.

[7] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, 1999.

[8] G. Chen and B. K. Szymanski. Four types of lookback. In *PADS '03: Proceedings of the seventeenth workshop on Parallel and distributed simulation*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

[9] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. Gtw: a time warp system for shared memory multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pages 1332–1339, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[10] M. Frank. *Reversibility for Efficient Computing*. PhD thesis, University of Florida, 1999.

[11] R. M. Fujimoto. Performance of time warp under synthetic workloads, January 1990.

[12] J. George B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1):50–87, 1986.

[13] F. Gomes. *Optimizing Incremental State-Saving and Restoration*. PhD thesis, University of Calgary, 1996.

[14] A. Griewank, D. Juedes, and J. Utke. Algorithm 755: Adolc: a package for the automatic differentiation of algorithms written in c/c++. *ACM Trans. Math. Softw.*, 22(2):131–167, 1996.

[15] L. P. J. Grimm and N. Rostiang-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. Technical report, nstitute National de Recherche en Informatique et en Automatique (INRIA), 1996.

[16] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[18] R. Landauer. Irreversibility and heat generation in the computing process. In *IBM Journal of Research and Development*, volume 5, pages 183–191, 1961.

[19] P. L'Ecuyer and T. H. Andres. A random number generator based on the combination of four lcgs. *Math. Comput. Simul.*, 44(1):99–107, 1997.

[20] L. Li and C. Tropper. Event reconstruction in time warp. In *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation*, pages 37–44, New York, NY, USA, 2004. ACM Press.

[21] Y.-B. Lin and B. R. Preiss. Optimal memory management for time warp parallel simulation. *ACM Trans. Model. Comput. Simul.*, 1(4):283–307, 1991.

[22] Y.-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in time warp simulation. In *PADS '93: Proceedings of the seventh workshop on Parallel and distributed simulation*, pages 3–10, New York, NY, USA, 1993. ACM Press.

[23] K. Perumalla and R. Fujimoto. Using reverse circuit execution for efficient parallel simulation of logic circuits. volume 4793, pages 267–275. SPIE, 2003.

[24] K. S. Perumalla. μsik : A micro-kernel for parallel/distributed simulation systems. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 59–68, Washington, DC, USA, 2005. IEEE Computer Society.

[25] F. Quaglia. Fast-software-checkpointing in optimistic simulation: embedding state saving into the event routine instructions. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 118–125, Washington, DC, USA, 1999. IEEE Computer Society.

[26] Y. Tang, K. S. Perumalla, R. M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko. Optimistic simulations of physical systems using reverse computation. *Simulation*, 82(1):61–73, 2006.

[27] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[28] D. West. Optimizing time warp: lazy rollback and lazy reevaluation. Master's thesis, University of Calgary, 1988.

[29] G. Yaun, C. D. Carothers, and S. Kalyanaraman. Large-scale tcp models using optimistic parallel simulation. In *PADS '03: Proceedings of the seventeenth workshop on Parallel and distributed simulation*, page 153, Washington, DC, USA, 2003. IEEE Computer Society.

[30] G. R. Yaun, D. Bauer, H. L. Bhutada, C. D. Carothers, M. Yuksel, and S. Kalyanaraman. Large-scale network simulation techniques: examples of tcp and ospf models. *SIGCOMM Comput. Commun. Rev.*, 33(3):27–41, 2003.

[31] Y. Zeng, W. Cai, and S. J. Turner. Batch based cancellation: a rollback optimal cancellation scheme in time warp simulations. In *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation*, pages 78–86, New York, NY, USA, 2004. ACM Press.