Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

MULTIPLEXED STATE SAVING FOR BOUNDED ROLLBACK

Fabian Gomes
Brian Unger

Department of Computer Science
The University of Calgary
Calgary, Alberta T2N 1N4, CANADA

John Cleary
Steve Franks

Department of Computer Science
University of Waikato
Hamilton, NEW ZEALAND

ABSTRACT

Optimistic parallel discrete event simulation (PDES) uses a state history trail to support rollback. State saving strategies range from making a complete copy of a model's state after each event execution to recording a sequence of each state modification made during event execution. The former is called copy state saving (CSS) and the latter incremental state saving (ISS). Periodic State Saving (PSS) a variant of CSS, saves the entire state but not after every event.

This paper presents a scheme for maintaining a multiplexed state history stream that interleaves a PSS mechanism with an optimized ISS mechanism. This multiplexed state saving (MSS) mechanism keeps forward execution overhead low while bounding rollback overhead over an arbitrary rollback distance. A key advantage of MSS over PSS is that events are not re-executed during a coasting forward phase.

Bounding rollback costs has two significant benefits. First, rollback delay can be controlled reducing rollback cascading and potential rollback thrashing. Second, interactive optimistic simulation with bounded response times to human queries can be supported. Absolute forward execution overheads for CSS and several ISS schemes are presented for two hardware platforms. A rough comparative analysis of MSS versus PSS and ISS is also included.

1 INTRODUCTION

State saving is one of the highest overheads associated with optimistic PDES. These overheads can severely limit the performance gains achieved through parallel execution. Numerous mechanisms have been developed to address the state saving problem including those by Bauer, and Sporer (1993), Bellenot (1992), Bruce (1995), Cleary et al. (1994), Gomes (1996), Gomes et al. (1996), Lin, and Lazowska (1990), Palaniswamy, and Wilsey (1993), Preiss et al. (1994), Rōnngren, and Ayani (1994), Rōnngren et al. (1996), Steinman (1993), and West, and Panesar (1996). This paper presents a new scheme that minimizes both forward execution overhead and rollback overhead.

Here we examine state saving within a logical process modeling methodology. The real world system being studied is viewed as a network of interacting physical components. A distributed simulation model is realized by a corresponding system of computational units called logical processes (LPs). Each LP model incorporates a disjoint part of the system's state and LPs interact solely by sending and receiving timestamped messages. Receipt of a message corresponds to an event that occurs at the time specified in the timestamp.

Concurrent parallel execution of a simulation is achieved through an asynchronous execution of these LPs on multiple processors. Fundamental to real world systems is that the future cannot affect the past, called the causality constraint. Causality between events is ensured by an LP synchronization algorithm. The Time Warp protocol, based on the virtual time paradigm, is an optimistic synchronization algorithm in which LPs execute speculatively advancing only their own local virtual time (LVT).

Time Warp, innately non-blocking, does not strictly adhere to the local causality constraint while executing events. Rather, it detects potential synchronization errors and uses a rollback mechanism to recover from these potential errors. To enable rollback, Time Warp must maintain a history of all computation including past state history, messages received, messages sent and other rollback sensitive computation like dynamic memory management.

A synchronization error is detected when a straggler message arrives that may modify the LP's past state. Recovery involves rolling back the LP to a state that existed prior to the synchronization error. Memory reclamation of past history, also known as fossil collection, is implemented using a global control mechanism based on a global virtual time (GVT).

Maintenance of an LPs state history can easily become the dominant overhead during forward computation. Secondly, the overhead incurred in recovering the past state during rollback can cause significant delays resulting in rollbacks in other LPs and possibly a cascading avalanche of rollbacks. Both of these sources of overhead can severely limit the performance of optimistic systems. The multiplexed state saving (MSS) mechanism described here addresses both of these issues.

This paper is organized as follows. First, performance issues of the two basic state saving strategies are discussed in Sections 2 and 3. Performance limitations and possible optimizations are discussed for each of these strategies. In Section 4, the SimKit simulation system is described and the performance of CSS and several variations of ISS are presented. The MSS mechanism is then presented in Section 5 with a comparative performance analysis in Section 6. Conclusions appear in Section 7.

2 CHECKPOINTING STRATEGIES

Checkpointing the entire state of a simulation before (or after) each event has been referred to as Copy State Saving (CSS). Every event is a potential recovery point. Rollback to a recovery point requires finding the associated checkpointed state and copying it back. The performance issues associated with CSS and PSS are described in the following.

2.1 Copy State Saving (CSS)

The main performance issues associated with CSS can be discussed in terms of the state size, state structure, state fragmentation and state dynamics.

State size: For large state sizes and small compute grain per event, checkpointing cost can easily dominate simulation execution time. Perhaps the biggest problem with using CSS is that it is not robust. It is easy for programmers to inadvertently create very large states and their concomitant large checkpointing cost. Another problem with CSS is that it can consume large amounts of memory. Memory usage is proportional to the state size and number of copies that have been kept after GVT.

State structure: If it is important that the state be restored in the same location then the entire saved checkpoint needs to be copied back on rollback. If the state is addressed indirectly through a pointer then a pointer swizzling optimization may be used for state restoration. However, in languages which allow anonymous pointers it is very difficult to sustain this optimization because there may be pointers from one part of the state to another.

State fragmentation: When an LP's state resides in a single contiguous block of memory, it is possible on many architectures to highly optimize the checkpointing of large blocks by using the `memcpy()` system call on some UNIX systems. However, if the LP's state is distributed across memory (not contiguous), then each contiguous block would need to be individually checkpointed and restored.

State dynamics: Memory to be used for checkpointing can be pre-allocated when the size of an LP's state is known. However, if dynamic state allocation is allowed then memory pre-allocation optimization can no longer be possible. Dynamic state also causes state fragmentation.

The two main disadvantages of CSS are: (i) large memory requirements that may even prevent the simulation from ever completing, and (ii) processing overheads which often significantly affect simulation performance as all events including those on the critical path, need to be checkpointed.

2.2 Periodic State Saving (PSS)

Bellenot (1992), Lin and Lazowska (1990), Palaniswamy and Wilsey (1993), Preiss et al. (1994), and Rōnngren, and Ayani (1994) show how PSS can overcome the shortcomings of CSS by checkpointing less frequently. State is no longer checkpointed for every event, rather, checkpoints are taken periodically by skipping a few events. The number of events skipped is termed as the *checkpoint interval.*

In PSS, it is quite probable that there may not be a checkpoint retained at a desired rollback point. An LP would have to roll further back in virtual time, to an event where a checkpoint was taken. Having restored the previous checkpoint, the state at the rollback point is *recomputed* by re-executing the events up to the rollback point. The LP is said to be in a *coasting forward phase* while re-executing these events. In the coasting forward phase, no output messages are resent. Fossil collection of events in PSS must take into account the need for retaining the single oldest checkpoint prior to GVT and subsequent input messages, to enable a rollback to GVT.

Real world simulation models tend to be non-homogeneous wherein LPs are not identical and tend to exhibit dynamic state referencing characteristics. In such situations there is no optimal fixed checkpoint interval. Preiss et al. (1994), and Rōnngren, and Ayani (1994) have proposed algorithms that use the recent execution information to estimate the checkpoint interval. The cost to dynamically estimate the checkpoint interval adds to the forward execution cost.

Perhaps the biggest disadvantage of periodic state saving is that on an average $N-1/2$ events need to be re-executed, where N is the checkpoint interval. The delay in coasting forward may be beneficial to other LPs local to the processor. The arrival of messages during this delay can enable better local scheduling decisions. However, LPs on remote processors are likely to perform speculative computations that will later be rolled back. Preiss et al. (1994) have noted that the latter cascading rollbacks cause a thrashing phenomenon which becomes dominant as the checkpoint interval increases.

Another serious disadvantage to PSS is that the rollback distance (i.e., the number of events invalidated during a rollback) is often small. Our experience with broadband network simulations show a large percentage of rollbacks that are fewer than 5 events. PSS is inefficient in such situations since either the checkpoint interval must be very small or a large number of events will need to be re-executed.

3- INCREMENTAL STATE SAVING (ISS)

Bauer, and Sporer (1993), Bruce (1995), Cleary et al. (1994), and Steinman (1993) have presented incremental strategies that only save modifications to state done in an event. A simple mechanism is back-trailing where, prior to a word being modified, its old value and address are saved in a linked list called backtrail. During state restoration, the backtrail is traversed in the reverse order, from the most recent event to the rolled back event, by writing the old values back into the associated addresses.

3.1- Performance

The major advantage of ISS is that both its execution and memory cost are simply related to forward event execution. The number of operations which involve modification to the LP's state tend to be a small stable fraction of all operations. For example, with checkpointing strategies, it is possible to dramatically increase the cost of state saving by something as simple as adding a large array to the state. In contrast any significant increase in ISS overheads is likely to be accompanied by a corresponding increase in the total event computation. Hence, ISS performance is only dependent upon the amount of state changed and not on the state size. This makes ISS a robust state saving mechanism. Furthermore, unlike checkpointing strategies, it is independent of state characteristics like structure, fragmentation and dynamics.

In ISS, every state modification saved during forward computation is undone during rollback. Multiple versions of the same state modified in an event, or over a sequence of events, may be restored redundantly. However, for state recovery only the earliest (since the rollback point) saved modification of a state variable needs to be restored. The rollback cost is proportional to the rollback length and is again a small fraction of the total cost that would be incurred if rolled back events had to be re-executed.

3.2- Transparency

A major disadvantage of ISS has been the need to insert state saving operations within LP model code. This requires great care on the programmer's part to insure that a state saving operation is inserted for every state change. Programming errors here can introduce very subtle bugs that are very difficult to find. This issue of transparency has been recently addressed by Gomes (1996), Gomes et al. (1996), Rōnngren et al. (1996), and West, and Panesar (1996).

Gomes et al. (1996), and Rōnngren et al. (1996) use parameterized data types to achieve transparent incremental state saving. Two templates are provided for declaring the state at the level of basic data types and pointer data types in a type safe way. In another approach, two new storage-specifier keywords "recover" and "nonrecover" were provided in the C^{++} language to facilitate the declaration of the simulation state. A C^{++} compiler front end was used to translate declarations using the storage specifier keywords into C^{++} code that uses the two templates.

Gomes (1996) addresses both the performance and transparency issue of ISS. Several optimizations of ISS that rely on static data flow analysis of the event processing routines are described. The optimizations attempt to decrease the amount of information saved in the backtrail. Moreover, a smart backtrail is constructed where a saved record identifies a restore code segment that needs to be executed on rollback. A saved record may contain additional parameters that are used by the restore code segment to reconstruct the state. Rollback performance was further improved by implementing a threading of the restore code segments. These optimizations can be incorporated in a compiler. While this technique requires all event processing source code to be available, West, and Panesar (1996) propose another technique that edits assembled code and inserts ISS calls automatically.

4- ISS IN SIMKIT

Gomes et al. (1995) present SimKit – a C^{++} class library that provides a simple elegant logical process view of simulation enabling both sequential and parallel execution without code changes to the application models. For parallel execution, SimKit uses the WarpKit Kernel, an implementation of Time Warp optimized for shared memory multiprocessors. The

WarpKit interface comprises of three classes, (i) `wk_simulation` - a run-time interface to the kernel, (ii) `wk_lp` - base class to derive simulation model's LP objects, and (iii) `wk_event` - base class for messages exchanged between LPs.

The Time Warp related quasi-operating services like *event-delivery*, *rollback* and *commit* are provided as virtual functions in the `wk_lp` class. These three functions are implemented in SimKit as follows: event-delivery may do any pre event initialization (for instance checkpointing) prior invoking the application model's event processing routine via the `sk_lp::process()` virtual function. The rollback interface is used to implement state restoration, and other rollback sensitive libraries like dynamic memory management. Fossil collection and commitment of irrevocable actions like output are implemented within commit. The SimKit application programmer's interface does not expose Time Warp related interface to the programmer. Only the declaration of state variables is necessary for ISS.

4.1 ISS Backtrail

The backtrail is a linked list of buffer blocks called State Log Blocks (SLBs). SLBs provide the buffer space to save records of state modifications. In the simplest form, a record also called the State Log Entry (SLE), is an old image of the state consisting of an address and value field. A free pool of SLBs is maintained on each processor.

An LP manages its own backtrail using four pointers, namely, `curSLB`, `curSLE`, `trail`, and `lastSLE`. `curSLB` and `trail` identifies the head and tail of the backtrail, while `curSLE` points to the most recent SLE recorded. `lastSLE` marks the end of the SLB and is used for SLB overflow handling. Each event has two pointers `evSLB` and `evSLE`. Each event demarcates the start of the state modification history (epoch) by saving the `curSLB` and `curSLE` pointers in `evSLB` and `evSLE`, respectively, prior to its execution.

Prior to a state modification, its image is saved by recording an SLE in the backtrail using an inlined `SaveState()`. The `lastSLE` in each SLB structure is initialized once to the last SLE in the array of SLEs when the free pool of SLBs are created.

```
struct SLB {
  SLB * nextSLB;
  SLB * lastSLE;
  SLE SLE_Array[SLB_SZ];
};

void SaveState( StAddress ) {
   /* tempSLB is a temporary variable */
  if ( curSLE != lastSLE )
    curSLE ++;
  else {␣ /* SLB overflows */
     /* Get a new SLB from pool */
    tempSLB = Processor->getSLB( );
     /* Link into backtrail */
    tempSLB->nextSLB = curSLB;
    curSLB = tempSLB;
     /* Reset overflow condition */
    lastSLE = curSLB->lastSLE;
     /* Initialize for next log entry */
    curSLE = curSLB->SLE_Array;
  }
   /* curSLE points to available space */
   /* Record state image */
  curSLE->address = StAddress;
  curSLE->oldValue = * StAddress;
}
```

Rollback, given the earliest event to be rolled back in the processed event history, is implemented by a backtrailing mechanism. Each SLE recorded from `curSLE` up to (not inclusive) the one identified by the `evSLE` field in the earliest event is restored by writing the old values into the associated addresses. SLBs that have been completely restored are reclaimed into the free pool. At the end of rollback, the LP's `curSLB` and `curSLE` point to the earliest rolled back event's `evSLB` and `evSLE`, respectively. The LP's `lastSLE` is initialized to `curSLB->lastSLE`. Commit involves reclaiming all SLBs from `trail` upto (not inclusive) the one identified by `evSLB` in the latest event not committed.

Variation of ISS, include saving a block of state variables in a variable length SLE structure. An optimization to variable length ISS is to prevent multiple SLEs from being recorded when a state block is modified multiple times in an event. Moreover, at rollback only the earliest saved SLE for a state block needs to be restored. This optimization is sustained by saving a timestamp of the event in the SLE and linking all SLEs for a given state block in the backtrail.

4.2 State Saving Overhead

The overhead for three variations of ISS that were implemented in SimKit are listed in Table 1 for two shared memory multiprocessor platforms, the SGI Power Challenge (64 bit, 75MHz processor) and the SPARC Server 1000 (32 bit, 50MHz processor). A simple all overhead application consisting of an LP (on a single processor) that repeatedly sends a message to itself is used to perform the measurements. The LP's state consists of 800 bytes of contiguous memory. The average cost to save 8 bytes was measured for each of the ISS variations: (i) ISS-8, one SLE saved per 8 byte state modified, (ii) ISS-V, one SLE saved per event, and (iii) BSS (Block State Saving), one SLE saved per event where each variable is saved at most once even if its modified multiple times during the event. CSS costs were measured using `memcpy()` to checkpoint the entire state of 800 bytes. The CSS cost shown in Table 1 lists the cost per 8-byte store when 800 bytes are checkpointed.

Table 1: Cost to Save 8 Bytes (in microseconds)

Platform	CSS	ISS-4	ISS-8	ISS-V	BSS
SGI	0.028	0.303	0.095	0.036	0.122
SPARC	0.201	0.538	0.318	0.314	0.244

These variations of ISS give a range of options for creating an efficient backtrail mechanism. A semantics where a state variable may be saved by either ISS variation requires the order in which the SLEs were recorded to be retained for state reconstruction on rollback. A simple solution would be to have a single unified backtrail using an additional ISS variation type field in each SLE. Alternately, each SLE record could hold the address of a restore code segment. One restore code segment per ISS variation would be needed where the last instruction in the restore code segment would effect a jump to the label identified in the next SLE while backtrailing.

This threading of restore code segments is used by Gomes (1996) to implement smart backtrails. A label is used to identify an inverse operation that uses parameters in the saved SLE for regenerating state or functions that effect end of SLB, end of rollback, etc. optimizations. These smart backtrails with optimizations based on data flow analysis of state modification in each event have been shown to decrease the forward state saving overheads from 38% to 8% of the average application level event processing time in a large broadband network application.

5 MULTIPLEXED STATE SAVING (MSS)

The multiplexed state saving mechanism combines the advantages of PSS and ISS. First, the motivation for MSS is outlined, and then a design is presented.

5.1 Motivation

The number of events rolled back, i.e. rollback distance, has significant performance implications for both PSS and ISS mechanisms. Short rollback distances in PSS either require a large number of events to be re-executed or a small checkpoint interval, thereby limiting the benefits of PSS. On the other hand, large rollback distances require a larger number of SLE records to be restored in ISS. Large rollback delays may result in a cascading avalanche of rollbacks and unstable Time Warp performance. One of the reasons for interleaving ISS and PSS is to reduce rollback delays and consequently possible cascading rollbacks and thrashing. A second need to bound rollback cost is to enable interactive input and output in Time Warp. The principle behind MSS is based on the following observations.

Observation 1. *The CPU cycles for restoring an LP's state by backtrailing is likely to be less than the cost for regenerating the state by re-executing the events.* In ISS, the cost to backtrail state modified in an event is a small fraction of the cost to execute the event.

Observation 2. *For situations where the checkpoint interval is large and the rollback distance is small, the number of events that would need to be re-executed in coasting forward phase would be larger than the number of events rolled back.*

Observation 3. *An LPs state can be reconstructed at a rollback point given a state modification history that led to an image of the LP's state.*

5.2 Design

The backtrailing mechanism presented in section 4 is used to implement the interleaved ISS. A buffer for checkpointing, referred to as `copyAhead` buffer is set aside. The LP maintains a pointer `copyAheadPtr` to this set aside buffer. Each event message has an additional pointer field called the `evCopyAheadPtr` which is set to `copyAheadPtr` prior to execution of the event. In addition, a pointer to the linked list of checkpoints is maintained by the LP using `chkptTrail`. Checkpointing is done as follows:

```
struct copyAhead {
  copyAhead * nextCopyAhead;
  SLE * ISSTrail;
  char buffer[CHKPT_SIZE];
};

void Checkpoint( ) {
  copy LPState into copyAheadPtr->buffer
   /* Link into checkpoint trail */
  copyAheadPtr->nextCopyAhead = chkptTrail;
   /* Drop a pointer into ISS trail */
  copyAheadPtr->ISSTrail = curSLE;
   /* reserve another copyAhead buffer */
  copyAheadPtr = Processor->getCopyAhead();
}
```

When rollback occurs, the `evCopyAheadPtr` in the earliest event rolled back is used to identify the *earli-*

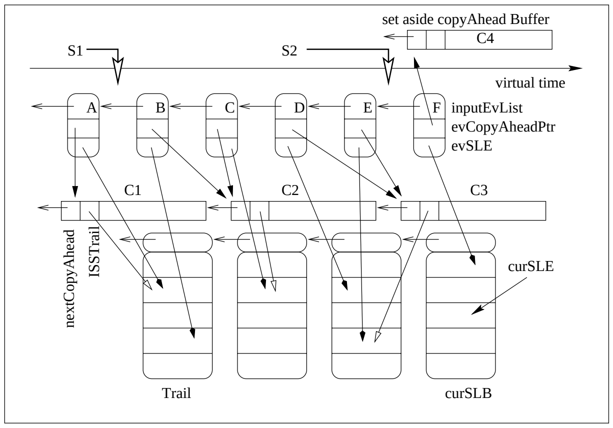

Figure 1: MSS State History Snapshot

est checkpoint saved (if any) since the rollback point. The following two cases may arise:

Case 1.- *A checkpoint exists since the earliest rolled back event.*
The `copyAhead` buffer identified by the rolled back event's `evCopyAheadPtr` contains the earliest checkpoint since the rollback point. After recopying the state, the state at the rollback point is restored by backtrailing from the SLE pointed to by the `ISSTrail` in the just used `copyAhead` buffer upto the SLE (not inclusive) identified by `evSLE` field of the earliest rolled back event.

Special care must be taken while freeing `copyAhead` buffers on rollback. The `copyAhead` buffer identified by the rolled back event's `evCopyAheadPtr` must be set aside instead of the previous set aside `copyAhead`. All buffers used for subsequent checkpoints as well as the previously set aside one can be freed. Similarly at fossil collection, the `copyAhead` buffer pointed to by the latest event that is being committed is retained in the linked list checkpoints, all earlier ones can be freed.

Case 2.- *No checkpoint was taken since the earliest rolled back event.*
This case is detected when the earliest rolled back event's `evCopyAheadPtr` points to the LP's `copyAheadPtr`. State restoration involves a simple backtrail as discussed in section 4.

A snapshot of the state history in MSS is sketched in Figure 1. Events `A`, `C` and `E` have taken snapshots. The `ISSTrail` pointer in each of the `copyAhead` buffer points to an SLE in the backtrail. The backtrail is made up of 4 SLBs where each SLB consists of an array of 5 SLEs. the arrival of a straggler `S1` would require checkpoint `C2` to be copied followed by 3 SLEs to be backtrailed, starting from `C2->ISSTrail` (inclusive) upto `B->evSLE` (not inclusive). `C3` and `C4` would be freed and `C2` would be the new set aside buffer. Arrival of straggler `S2` would result in 2 SLEs to be backtrailed starting from `curSLE`.

A possible optimization of MSS is that events that are checkpointed need not be incrementally state saved. To implement this optimization, two versions of every event will have to be coded. Alternatively, events that modify a large percentage of state may be candidates for checkpointing. This corresponds to application model based checkpointing rather than an automatic checkpointing where the interval is set dynamically or statically.

6- PERFORMANCE COMPARISON

An analysis of the relative average performance of MSS, PSS and ISS follows. The following notation is used in the analysis:
F_p – Forward execution cost for PSS per event,
F_i – Forward cost for ISS per event,
F_m – Forward cost for MSS per event,
R_p,R_i,R_m – Rollback cost for PSS, ISS and MSS,
S – State size in words,
α – Average % of state modified per event,
t_e – Average time to execute an event (model level),
t_m – `memcpy()` time per word,
t_w – 8byte word copy time,
N_p – Number of events in checkpoint interval for PSS,
N_m – Number of events in checkpoint interval for MSS,
D – Rollback distance in number of events.

Forward execution costs for PSS, ISS, and MSS are as follows:

$$F_p = \frac{St_m}{N_p}$$

$$F_i = \alpha St_w$$

$$F_m = \frac{St_m}{N_m} + \alpha St_w$$

Notice that $F_i < F_p$ when

$$\alpha < \frac{t_m}{t_w N_p} \approx \frac{0.7}{N_p}$$

and $F_m < F_p$ when

$$N_p < \frac{N_m}{1 + \alpha N_m (t_w/t_m)} \approx \frac{N_m}{1 + 1.4\alpha N_m}$$

$N_p < 1/(1.4\alpha)$ for large N_m (e.g. $N_p < 7$ if $\alpha = 10\%$) For this case,

$$\frac{F_m}{F_i} = \frac{t_m/t_w}{\alpha N_m} + 1 = \frac{7}{N_m} + 1.$$

Thus ISS has a lower forward execution overhead than PSS when α is less than $(100(t_m/t_w)/N_p)$, or equivalently, when N_p is less than $(t_m/t_w) \approx 1$. For this case MSS also has lower forward overhead than PSS.

The rollback overhead however favors MSS as follows.

$$R_i = \alpha S t_w D$$

$$R_p = S t_m + \frac{t_e N_p}{2}$$

$$R_m = \alpha S t_w D \quad if \quad D < (N_m/2)$$

$$R_m = S t_m + (\alpha S t_w N_m/2) \quad if \quad D > (N_m/2)$$

thus $R_m = R_i$ when $D < (N_m/2)$ and
$R_m < R_i$ when $D > (t_m/\alpha t_w) + (N_m/2)$.

For example, when $t_m/t_w = 0.7$ and $\alpha = 10\%$ then MSS has a lower rollback overhead than ISS if $D > 7 + N_m/2$, and of course MSS is equivalent to ISS when $D < N_m/2$. Comparing R_m and R_p gives $R_m < R_p$ when

$$S t_m + \frac{\alpha S t_w N_m}{2} < \frac{S t_m}{+} \frac{t_e N_p}{2}$$

or when

$$t_e > \frac{\alpha S t_w N_m}{N_p}.$$

The event execution time t_e must include calculations that incur at least $\alpha S t_w$ since on average αS amount of state is modified during the event. Each such state modification is at a cost of about t_w. If $t_e > 2\alpha S t_w$ then MSS performs better than PSS when $N_m < 2N_p$.

7 CONCLUSIONS

Except for unusual cases where (i) there is either a small amount of state information or a large amount of state modified per event, or (ii) there is unacceptable rollback overhead due to very large checkpoint intervals, ISS will have the lowest overhead during forward computation than any of the other approaches, i.e., MSS, PSS and CSS. Since the critical path of any parallel execution incurs this forward overhead, ISS would appear to have relatively robust superior performance during forward computation.

However, the forward overhead of MSS can be made arbitrarily close to that of ISS by increasing the checkpointing interval. This is feasible since, in general, longer checkpointing intervals can be used with MSS than with PSS. Further, MSS can outperform PSS in the forward direction since larger checkpoint intervals can be used while incurring similar rollback overheads.

On the other hand, MSS has bounded rollback overhead that is typically lower that ISS, PSS or CSS. This enables limiting rollback delays and thus controlling the potential for cascading rollbacks. In addition, MSS has been used by Franks et al. (1997) to support interactive input and output in optimistic systems where the response time to human queries, for example, needs to be bounded.

Our experience with one large industrial strength application, a broadband ATM simulation, suggests that there are many rollbacks with very short distances, and occasional rollbacks with very large distances. We can only conjecture that the behavior of MSS will provide robust high performance optimistic execution.

Future work is required to determine whether MSS can deliver better overall Time Warp performance than ISS. Exploring the relative advantages of MSS and ISS requires further experimentation with a range of applications that exhibit varied rollback distance behavior.

REFERENCES

Bauer, H., and C. Sporer. 1993. Reducing rollback overhead in Time Warp based distributed simulation with optimized incremental state saving. In *Proceedings of the 26th Annual Simulation Symposium*, ed. J. Miller, 12–20. Arlington, Virginia.

Bellenot, S. 1992. State skipping performance with the Time Warp operating system. In *Proceedings of the 1992 SCS Western Simulation Conference on Parallel and Distributed Simulation*, eds. M. Abrams and P. Reynolds Jr., 53–64. Newport Beach, California.

Bruce, D. 1995. Treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS95)*, ed. Y-B. Lin, and M. Bailey, 40–49. Lake Placid, New York.

Cleary, J., F. Gomes, B. Unger, X. Zhonge, and R. Thudt. 1994. Cost of state saving and rollback. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94)*, ed. D. Arvind, R. Bagrodia, and Y-B. Lin, 24(1):94–101. Edinburgh, Scotland, U.K.

Franks, S., F. Gomes, B. Unger, and J. Cleary. 1997. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS97)*, eds. R. Ayani and C. Tropper, Burg Lockenhaus, Austria.

Gomes, F. 1996. Optimizing Incremental State Saving and Restoration. *PhD thesis*, Department of Computer Science, University of Calgary, Canada.

Gomes, F., J. Cleary, and B. Unger. 1996 Language based state saving extensions for optimistic parallel simulation. In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, California.

Gomes, F., S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington. 1995. SimKit: A high performance logical process simulation class library in C^{++}. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos, K. Kang, W. Lilegdon, and D. Goldsman, 706–713. Arlington, Virginia.

Lin, Y-B. and E. Lazowska. 1990. Reducing the state saving overhead for Time Warp parallel simulation. *Technical Report 90–01–02*, Department of Computer Science, University of Washington, Seattle, Washington.

Palaniswamy, A. and A. Wilsey. 1993. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93)*, eds. R. Bagrodia and D. Jefferson, 23(1), 127–134. San Diego, California.

Preiss, B., W. Loucks, and I. MacIntyre. 1994. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253.

Rōnngren, R., and R. Ayani. 1994. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94)*, ed. D. Arvind, R. Bagrodia, and Y-B. Lin, 24(1):110–117. Edinburgh, Scotland, U.K.

Rōnngren, R., M. Liljenstam, R. Ayani, and J. Montagnat. 1996. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS96)*, 70–77. Philadelphia, PA.

Steinman, J. 1993. Incremental state saving in SPEEDES using C^{++}. In *Proceedings of the 1993 Winter Simulation Conference*, 687–696. Los Angeles, California.

West, D., and K. Panesar. 1996. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS96)*, 78–85. Philadelphia, PA.

AUTHOR BIOGRAPHIES

FABIAN GOMES is a post doctorate fellow in the Dept. of Computer Science at the University of Calgary. He received his Ph.D. degree in Computer Science from the University of Calgary in 1996. His research interests are in parallel simulation, modeling ATM broadband networks, distributed systems and rollback based computing.

BRIAN UNGER is a professor in the Dept. of Computer Science at the University of Calgary. His current interests include the modeling and simulation of ATM broadband networks, parallel simulation and simulation in Java.

JOHN CLEARY is a professor in the Dept. of Computer Science at Waikato University. His research interests include parallel and distributed systems, logic programming, complexity theory applied to adaptive and learning systems, and parallel simulation.

STEVE FRANKS is a lecturer in the Dept. of Computer Science at Waikato University. He is pursuing a Ph.D. degree in Computer Science at the University of Calgary. His research interests include interactive visual simulation, graphics and multimedia.