Just-In-Time Cloning

Maria Hybinette

Computer Science Department
University of Georgia
Athens, GA 30602-7404, USA

maria@cs.uga.edu

Abstract

In this work we focus on a new technique for making cloning of parallel simulations more efficient. Cloning provides a means for running multiple similar simulations in parallel where many computations are shared rather than repeated [12]. A simulation is cloned on an event for a particular set of logical processes, creating new LP clones. The clones diverge as messages from the new LPs arrive at uncloned LPs. Until replication, all the computations for a particular LP are shared between the clones. Simulation kernels using cloning achieve efficiency by incrementally replicating logical processes as necessary. This enables regions of the simulation that have not been affected to use the same computations for both the new and old clone. Clearly, the longer replication can be delayed, the more efficient the simulation. We hypothesize that in many cases replication takes place before it is strictly necessary. We propose **just-in-time cloning** *that addresses this issue by relaxing the constraints of simulation cloning to further benefit from shared computations.*

1. Introduction

In this work we focus on a new technique for making cloning of parallel simulations more efficient. We briefly review the cloning approach here. For a full description please see [12]. We report on a significant efficiency improvement to the algorithm for simulation cloning.

Simulation cloning in Time Warp simulations enables efficient exploration of multiple simulated futures based on policy decisions made at decision points. Time Warp simulations are composed of multiple logical processes (LP) that exchange messages to advance the simulation. In the standard approach to cloning, a simulation is duplicated a portion at a time by replicating LPs as necessary. Our improvement concerns the timing of LP replication. In particular, we are able to delay many replications and to avoid some replications altogether.

As an example of how simulation cloning might be applied, consider an air traffic control scenario where inclement weather may force a change in policy for one or more airports. One might use simulation to compare alternate strategies for managing the flow of aircraft in this portion of the air space. Many strategies are available to controllers to address such problems, but their impact on the situation may be difficult to predict. Simulation cloning can provide a means for efficiently evaluating many policies at once.

Continuing with the example, the simulation can be initialized with the current state of the airspace, then executed forward until reaching the point where new policies are to be imposed. The simulation can then be cloned (replicated), with each clone simulating the traffic space with a different control policy. The point where the simulation is cloned is referred to as a *decision point*. The cloned simulations execute concurrently, and will produce results identical to a traditional replicated simulation where the entire simulation is executed from the beginning using different policies.

Simulation cloning assumes a message-based computation paradigm like that commonly used in parallel discrete event simulations. Specifically, the simulation is composed of a collection of *logical processes* (LPs) that communicate exclusively by exchanging time stamped events or messages. A synchronization algorithm is used to ensure that each LP processes its events in time stamp order, or in the case of *optimistic* simulation protocols, the net effect of each LP's computation is as if its events were processed in time stamp order. Simulation cloning can be used with either conservative or optimistic synchronization techniques.

Simulation cloning improves performance because it allows computations to be shared between the clones. Up to the decision point the simulations are identical and a single computation suffices for the cloned simulations. After the decision point the simulations slowly diverge as aircraft emanating from the airport of interest begin to affect other airports in the simulation. In air traffic simulations airports are typically mapped to LPs[19]. As an airport (LP) accepts an incoming aircraft from an airport (LP) that diverged, it too diverges (or replicates). Eventually, the entire simulation is fully replicated and the simulations are doing all the computations as two replicated simulations plus some small overhead.

We envision two general schemes for improving the performance of cloning: (1) By **delaying** the replication of a cloned logical process until it is absolutely necessary, thus allowing LPs to continue to share computations and (2) by **merging** cloned simulations so that replicated simulations that have re-converged to the same state can resume sharing computations. This paper focuses on a discussion of delaying replication.

As an example of how delaying replication may be beneficial, consider a single flight from a replicated LP representing the Atlanta (ATL) airport to Stockholm (ARN) that is forced to land one hour late. Further assume that this late arrival has no additional effect on the state of the Stockholm airport. In standard simulation cloning, the Stockholm airport is replicated by the delayed flight since now there are two different simulations. The cloning of the Stockholm's airport is illustrated in Figure 1, where a new physical clone 2 is created and virtual clone 2 is remapped to Stockholm's physical clone 2. As aircraft depart from the *new* Stockholm airport clones destined to other European airports, those airports will also be cloned, even if the two copies of the Stockholm airport are equivalent. In Figure 1 a flight from Stockholm's clone 2, destined to Vienna, has an identical flight in clone 1 and will cause Vienna to clone. The two physical copies of Vienna will be the same after receiving both copies of the flights from Stockholm. The problem here is that by virtue of an interaction from a diverged clone, airports become unnecessarily replicated.

In this work we propose a *just-in-time cloning* as an approach to delaying replication of cloned logical processes. We evaluate the approach in two benchmark simulation applications: PCS and P-Hold.

2. Related work

Simulation cloning has been shown to improve performance of interactive applications such as simulations of the ground transportation [16, 17] and air-traffic control [12]. Cloning has been implemented both in conservative [2, 16, 17] and optimistic [12] simulation executives.

With respect to cloning in general, related work in interactive parallel simulation include [7] and [6]. The approach of [7] allows for the testing of what-if scenarios provided they are interjected before a deadline. Alternatives are examined sequentially using a rollback mechanism to erase

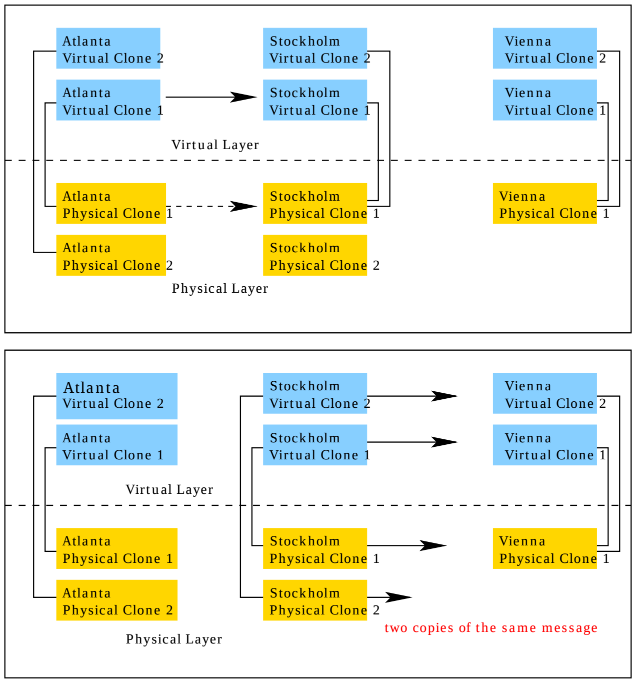

Figure 1. Sequence of Events in Standard Simulation Cloning: *Top:* **A delayed flight from Atlanta arriving at Stockholm forces Stockholm to clone physical clone 2;** *Bottom*: **A flight, from Stockholm's clone 1 has an identical flight in clone 2, causes Vienna to clone.**

exploration of one path in order to begin exploration of another. [6] create a new simulation from a previous base-line simulation by executing forward. They can determine if a computation can be reused by using a predicate function that is tested on the baseline simulation, however as in [7] only the testing of one alternative at a time is allowed. A drawback of this latter approach is that one must manage the entire state-space of the baseline simulation.

Cloning has been used to improve accuracy of simulation results, i.e., to run multiple independent replications then average their results at the end of the runs. For example, in [10] research focuses on how to achieve statistical efficiency of replications spread on different machines. In [18] replications are synchronized via a shared clock. In this way the same event occurs at the same time at all replicas.

Cloning is also used in sequential simulations to propagate faults in digital circuits [14], modeling of flexible manufacturing systems [4] or to fork transactions in simulation languages [11]. Sequential simulation languages typically clone dummies ("shadows") to do time-based waiting [15].

Our approach that delays replication is similar to techniques that reduce the cost of rollbacks: Lazy cancellation [9], the aggressive no-risk (ANR) [5] protocol and exploiting "lookback" [3] for example. Lazy cancellation delays sending anti-messages until the event that originally scheduled them is guaranteed not to be re-generated. ANR delays delivering messages until the event that created them is guaranteed not to be rolled back (the sender event), this technique avoids cascading rollbacks. Lookback is a techniques that identifies situations where one can avoid rollback.

Our approach is applicable both to optimistic and conservative protocols. It uses message comparison as in the lazy cancellation protocol, but does not suffer from the same performance penalty since it only compares messages if the LP is subject to replication.

3. Problem statement

A Time Warp simulation is composed of multiple *logical processes* that communicate by exchanging messages. To support multiple parallel clones of a simulation, the concepts of *physical logical process*, *virtual logical process* and *virtual messages* were introduced in [12]. Virtual LPs and messages do not have a physical realization. Instead, each virtual LP maps to one *physical logical process* that does have a physical realization in the parallel simulation system. Similarly, each virtual message maps to a single physical message.

Computations common between two or more clones are completed by a physical LP and shared by the virtual LPs in the clones that are mapped to the physical LP. Similarly, virtual messages among different clones are mapped to a single physical message to avoid duplication. Sharing of common computations may proceed until the state of one of the virtual LPs diverges from the others. At this point the LP must be replicated and it becomes a physical LP.

We know that a virtual LP cannot diverge from the others until it receives a message from a replicated LP. It is possible at this point that the information in a message from a replicated LP sender will cause the state in the receiver to diverge from the others. An earlier implementation of cloning took a conservative approach, and replicated the receiving LP when this occurs [12]. However, if replication can be delayed we may be able to gain even more efficiency.

Observe that some replicated LPs generate messages identical to their un-replicated (pre-cloned) counterparts. In these cases it is not necessary to replicate LPs that receive such messages. If we account for these situations and delay replication there may be a significant advantage, especially considering that if we did clone identical logical processes they in turn would send out identical messages causing further unnecessary cloning and so on. This observation is the basis of our new approach, *just-in-time cloning*.

The two problems we must address then, are:

- How long can we delay replication? and
- How can we implement just-in-time cloning so that the resulting simulation outcome is correct?

4. Implementation

As long as messages for a virtual LP from a replicated LP are identically matched by messages from the other corresponding replicated LPs there is no need to replicate the receiver. Thus our implementation of just-in-time cloning relies on message checking to detect situations where replication is necessary.

Message comparison is expensive. If we do too much comparing, performance will suffer. However, it is not necessary that all messages be tested, it is only necessary for certain virtual LPs to conduct this comparison. We hypothesize that the set of LPs that must inspect messages is usually a small fraction of those in the entire simulation.

Consider the fact that simulations are cloned on an event at a specific LP. So, initially, only one LP is replicated. From that point, changes spread across the simulation outward, much like the ripples from a pebble dropped in a pond. It is at the boundary, the leading edge of the ripple, where messages must be checked. Now consider an LP on this boundary. It has received a message from a replicated LP. Must it replicate itself? Perhaps not. If it receives an identical message from all the other corresponding senders, it does not have to replicate. But how long should it wait to receive these messages?

We implemented just-in-time cloning in an optimistic simulator. In this case we know that it is probable that the corresponding messages have been received by the time GVT matches the timestamp of the first message, however this is not guaranteed[1]. So, at this time we check to see if all the messages have arrived; if not, the receiving LP is replicated. The net effect is that the LPs on the boundary execute events conservatively, while the rest of the simulation proceeds optimistically.

Just-in-time cloning was implemented within an optimistic Time Warp simulation kernel [13]. Below we provide pseudocode for the portions relevant to the JIT algorithm.

```
Data structures:
   // Each LP has an associated array of queues;
   // one queue for each simulation clone.  Queues
   // are sorted in timestamp order.
   delay_queue[LP][clone]

Event processing:
   // get next message
   message = dequeuePendingQueue();

   if( message might cause replication )
        {
        rcvt = message.receive_time

        // block messages after rcvt;
        // handled by TimeWarp executive
        sendBlockingMessage( self,rcvt,putInJITList )

        // queue message in delay queue
        enqueue( delayQueue[self][message.clone] );
        }
   else
        {
        process message normally
        }

Cancellation: // if there is a rollback
   remove message from delay queue
   if message scheduled a blocking message,
cancel it
```

We add a data structure, `delay_queue[][]`, which contains an array of queues for each LP. For each LP, the array contains one queue for each simulation clone. The queues are used to store incoming messages that might force replication.

When a new message arrives, it is inspected to see if it might cause replication. Specifically, if it is from a replicated LP, and the receiving LP has not yet been replicated, the message may cause a replication. If the message might cause replication the LP sends a `BlockingMessage` to itself (more on that in a moment), and adds the message to the appropriate `delay_queue`. If the message cannot cause replication it is processed normally.

[1]this only affects the performance and not the correctness of the simulation

The sending of a `BlockingMessage` with a timestamp of the incoming message sets the following chain of events in motion. First, the optimistic executive will block processing on all messages with a timestamp after the time of the `BlockingMessage`. This ensures that the messages equal to or larger than GVT and less than the time stamp of the `BlockingMessage` are available for processing and allows our software to capture the messages that need to be inspected.

Next, our callback function `PutInJITList` (see below) will be called automatically when GVT reaches the timestamp. `PutInJITList` and `ProcessJITLIST` accomplish evaluation of the `delay_queue`s to see if the LP must be replicated. Code for `PutInJITList` and `ProcessJITLIST` is included below.

Note that it is possible for one or more messages to still be in transit when the `delay_queue` is processed. Therefore, an LP may be unnecessarily replicated. This may lead to inefficient, duplicated processing by the new LP, but the simulation will still produce correct results.

```
PutInJITList() // called when GVT reaches blocktime
   JITList = NULL;
   Put LPs that sent blocking messages scheduled
   at GVT onto JITList
   ProcessJITList( GVT )

ProcessJITList( GVT )
 while( ((readyLP = next from JITList ) != NULL) )
    merged_message = NULL;
    for( clone = 0; clone < num_clones; clone ++ )
      {
      message = dequeueAtTime( GVT,
                     delayQueue[readyLP][clone] )
      if( message != NULL)
         if( merged_element == NULL )
            merged_message = message;
         else
           if( checkEqual( merged_message, message ) )
              mergeMessage( merged_message, message )
           else
               queue message in Pending Queue
      } /* for all clones */
    process merged_message
```

`dequeueAtTime()` will only return a pointer to a message if one or more exist at the specified time; in our case GVT.

We require `checkEqual()` to be provided by the application programmer. `checkEqual()` evaluates its two message arguments to determine if they are equivalent. We do this for two reasons. First, it may be the case that two messages are not bit-wise identical, but still equivalent. The kernel cannot determine this without help from the application level. Second, the user may consider that two very slightly differing messages may be considered identical. By relaxing the constraint on how similar messages must be to be considered identical, we may achieve more efficiency.

(The performance of this aspect of just-in-time cloning has not been evaluated yet.)

If the two messages are equivalent, they are merged by `mergeMessage()`. Our cloning implementation adds additional information to messages in order to determine the origin of the message (i.e. if it came from a replicated LP). `mergeMessage()` ensures that the data structure is appropriately updated when the messages are combined.

5. Performance

We evaluated the performance just-in-time cloning using two benchmark applications: P-Hold and PCS [8, 1]. For each application we constructed best case and worst case scenarios. In the best case scenarios, a single LP is replicated, but it continues identically to the original LP, thus additional replications do not usually occur. Unnecessary replications are possible due to the race condition outlined in Section 3, but they cause inefficiency at worst; not incorrect results. In the worst case scenarios, every message from a replicated LP to an un-replicated LP forces replication. The worst case was implemented by revising the application-level function `checkEqual()` to always return `false`.

The best and worst case scenarios help us explore the upper and lower performance bounds of just-in-time cloning for a particular application. For reference we also include results of traditional replication (two fully physically replicated simulations) and a single simulation. By "single simulation" we mean the original simulation has not been cloned.

P-Hold provides synthetic workloads using a fixed message population. Each LP is instantiated by an event. Upon instantiation, the LP schedules a new event with a specified time-stamp increment and destination LP (for more details on P-Hold see [8]). P-Hold is instrumented by varying the rate at which a clone causes the replication of other LPs. This is done by adjusting the selection of the destination LP of a message (logical processes are instantiated by receiving messages).

We instrumented to variations of P-Hold. In the first case the spreading is slowed by having an LP send messages only to itself or to its neighbor (slow spreading). In the second case the spreading is not constrained and the destination is selected from a uniform distribution of all logical processes in the simulated system (fast spreading). The time stamp increment to schedule a new event is 1.0 in the slow spreading case and is selected from an exponential distribution between 0.0 and 1.0 in the fast spreading case. In our experiments the P-Hold simulations use a message population of 8096 and 1024 logical processes.

PCS is a benchmark which simulates a personal communication services network. The network is wireless (radio receivers and transmitters) and provides services to mobile PCS subscribers. The radio ports are structured in a grid with one port per sector. Each cell services *portables* or mobile phone units that each occupy a cell for a period of time before proceeding to one of the four neighboring cells. Call arrival, call completion, and move are example behaviors or types of portables which are modeled by events in the simulation. Cells are modeled by LPs. Here we use a PCS network of 1024 cells (a $32X32$ grid) and 8192 portables. Most of the communication, typically 90%, is between LPs residing on the same processor. Many of those messages are self-initiating, or sent between the same logical process. More details of PCS are described in [1].

Evaluation of just-in-time cloning was conducted on an SGI Origin 2000 with sixteen 195 MHz MIPS R10,000 processors. The first level instruction and data caches are each 32 KB. The unified secondary cache is 4 MB. The main memory size is 4 GB. All experiments use 4 processors

The independent variable for this experiment is execution time and the metric is simulation progress (virtual time). Larger values indicate better performance. Simulated time is also referred as the virtual time to distinguish it from wall-clock time. For all P-Hold simulations the decision point (cloning) occurs at one logical process (LP_0) at simulated time 1000. In PCS the decision point is at simulated time $500, 000$.

Figure 2 shows the simulation progress (virtual time) of P-Hold where the destination address is selected from a uniform distribution (the fast spreading case) versus wall clock time.

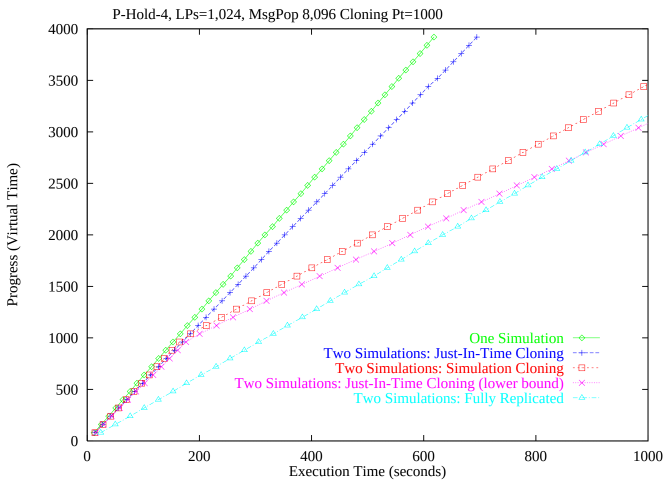

Figure 2. Performance of P-Hold Where destination LP Is Selected from a Uniform Distribution of All LPs in the Simulation. Larger numbers indicate better performance.

Figure 3 shows the performance of P-Hold in the slow spreading cases. In both plots a larger number indicates better performance.

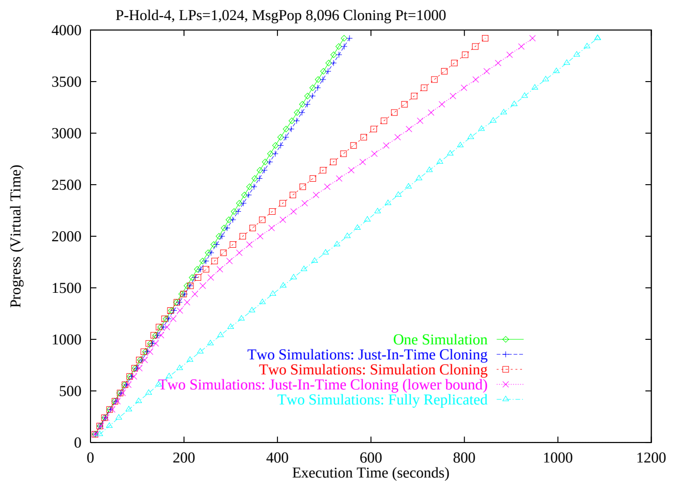

Figure 3. Performance of P-Hold Where destination LP Is Either Self or Nearest Neighbor. Larger numbers indicate better performance.

The performance for traditional cloning, in the fast spreading scenario, initially degrades upon the instantiation of a new clone, but then later continues linearly. The execution time for the best case of just-in-time cloning slows down but is closer to the performance of a single simulation. Just-in-time cloning, in the best case, improves on simulation cloning by 39%, and is about 12% slower than a single simulation. Its worst case performance is about 14% slower than simulation cloning.

In simulation cloning the slow spreading scenario shows a more dramatic slow-down but then proceeds linearly as well. The slow-spread case performs significantly faster than the fast-spread case, once the performance stabilizes. In this scenario, the performance of just-in-time cloning is tight to the performance of a single simulation, there is about about 2% difference. It improves on simulation cloning by about 34% and at its worst it is about 12% slower than simulation cloning.

The best case is tighter to a single simulation in the slow spread scenarios since fewer logical processes interact with a cloned logical process, causing them to wait conservatively for “corresponding” messages until they can progress further. In contrast in the fast spread scenario, there is a larger set of *neighboring* logical processes and hence more logical processes progress in conservative mode. The worst case of just-in-time cloning perform worse than simulation cloning because just about every logical process is likely to interact with the cloned logical process, causing all logical process to progress in conservative mode.

Our results, shown in Figure4 from the experiments using the PCS benchmark show similar performance as P-Hold, except in the worst case. The worst case, that always replicates a logical process when it receives a message from a cloned logical process initially behaves similar to the performance of simulation cloning but then at simulated time 1,750,000 it degrades more toward the performance of the fully replicated PCS simulation. The performance of the fully replicated PCS eventually surpasses the worst case performance of just-in-time cloning. In the best-case however, just-in-time cloning is closer to the performance of a single simulation.

Figure 4 shows the performance of PCS. These performance improvements of both PCS and P-Hold indicate that there may be a significant performance improvement of just-in-time cloning over simulation cloning which makes a difference for an interactive simulation, where alternatives need to be compared and instantiated dynamically.

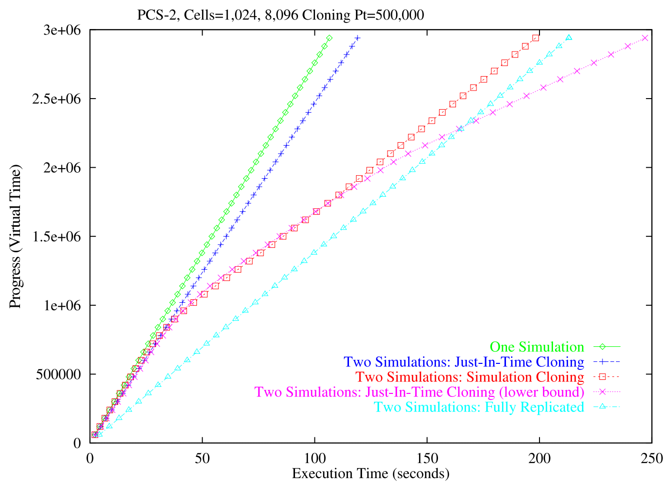

Figure 4. Performance of PCS. Larger numbers indicate better performance.

6. Conclusion

Just-in-time cloning enables cloned simulations to avoid replication of receiver LPs, thus enabling computations to be shared longer. The potential benefit of just-in-time cloning is demonstrated using two benchmarks, P-Hold and PCS. Results indicate that just-in-time cloning can improve the performance of simulation cloning by a factor of 1.7. It is especially efficient when the influence of a message introduced by a logical process does not spread to other processes in the simulation at a high rate. At best it runs only 2% slower than a *a single simulation* run. These initial performance results indicate that just-in-time cloning may significantly reduce the time required to compute multiple simulations.

At present we have only evaluated the approach in synthetic best and worst case scenarios. In future work we will investigate performance in typical simulation tasks. Additionally, we assume that the cost of message comparison is low. This may be reasonable since we only need to to compare messages when an LP receives messages from a cloned LP. In future work we will examine the impact of message size and also explore realistic applications for just-in-time cloning.

References

[1] C. D. Carothers, R. Fujimoto, Y.-B. Lin, and P. England. Distributed simulation of large-scale PCS networks. In *MASCOTS*, pages 2–6, 1994.

[2] D. Chen, S. J. Turner, B. P. Gan, W. Cai, J. Wei, and N. Julka. Alternative solutions for distributed simulation cloning. *Simulation: Transactions of The Society for Modeling and Simulation International*, 79:299–315, 2003.

[3] G. Chen and B. K. Szymanski. Four types of lookback. In *Proceedings of the seventeenth workshop on Parallel and distributed simulation*, page 3. IEEE Computer Society, 2003.

[4] W. J. Davis. On-line simulation: Need and evolving research requirements. In J. Banks, editor, *Handbook of simulation: Principles, methodology, advances, applications, and practice*. John Wiley & Sons, August 1998. Co-published by Engineering and Management Press.

[5] P. Dickens and P. Reynolds. SRADS with Local Rollback. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 161–164, 1990.

[6] S. L. Ferenci, R. M. Fujimoto, M. H. Ammar, and K. Perumalla. Updateable simulation of communication networks. In *Proceedings of the* 16^{th} *Workshop on Parallel and Distributed Simulation (PADS-2002)*, pages 107–114, May 2002.

[7] S. Franks, F. Gomes, B. Unger, and J. Cleary. State saving for interactive optimistic simulation. In *Proceedings of the* 11^{th} *Workshop on Parallel and Distributed Simulation (PADS-97)*, pages 72–79, 1997.

[8] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 23–28. SCS Simulation Series, January 1990.

[9] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *SCS Multiconference on Distributed Simulation*, volume 19, pages 61–67, February 1988.

[10] P. W. Glynn and P. Heidelberger. Analysis of parallel replicated simulations under a completion time constraint. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1(1):3–23, 1991.

[11] J. O. Henriksen. An introduction to SLX. In *Proceedings of the 1997 Winter Simulation Conference*, pages 559–566, December 1997.

[12] M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(4):378–407, 2001.

[13] D. R. Jefferson and H. Sowizral. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Technical Report N-1906-AF, RAND Corporation, December 1982.

[14] K. P. Lentz, E. S. Manolakos, E. Czeck, and J. Heller. Multiple experiment environments for testing. *Journal of Electronic Testing: Theory and Applications*, 11(3):247–262, December 1999.

[15] T. J. Schriber and D. T. Brunner. Inside discrete-event simulation software: How it works and why it matters. In *Proceedings of the 1997 Winter Simulation Conference*, pages 14–22, December 1997.

[16] T. Schulze, S. Strassburger, and U. Klein. On-line data processing in simulation models: New approaches and possibilities through hla. In *Proceedings of the 1999 Winter Simulation Conference*, pages 1602–1609, December 1999.

[17] T. Schulze, S. Strassburger, and U. Klein. Hla-federate reproduction procedures in public transportation federations. In *Proceedings of the 2000 Summer Computer Simulation Conference*, July 2000.

[18] P. Vakili. Massively parallel and distributed simulation of a class of discrete event systems: a different perspective. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 2(3):214–238, 1992.

[19] F. Wieland. Parallel simulation for aviation applications. In *Proceedings of the IEEE Winter Simulation Conference*, pages 1191–1198, December 1998.