Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

WOLF: A rollback algorithm for optimistic distributed simulation systems

Vijay Madisetti
Jean Walrand
David Messerschmitt

Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Discrete event dynamical systems are used to model a number of engineering applications ranging from communication networks, distributed computing systems to manufacturing systems. A new analytical model is proposed for the analysis of asynchronous, optimistic distributed simulation of discrete event dynamical systems. The performance of traditional timewarp algorithms and that of a new rollback algorithm, Wolf, are examined in this framework and a comparison is attempted. A few preliminary implementations of Wolf on a distributed computing system confirm analytical results predicting a considerable reduction in error propagation, a decrease in the number of cancellations and an enhancement in the forward computation.

1. INTRODUCTION

We propose to investigate the use of distributed simulation for the evaluation and design of discrete event dynamical systems(DEDS). DEDS are used in a gamut of applications ranging from communication networks, distributed computing systems to manufacturing systems. These models, nevertheless, are characterized by their complexity and the lack of suitable performance evaluation methods. Except for a very small set, these systems are analytically intractable, and are in addition prohibitive to evaluate numerically. As a consequence, one often has to resort to simulation as the only available and sufficiently general evaluation methodology. However, simulations of complex DEDS are usually exceedingly slow to run (and also to develop). As a result, the development of simulation speedup methods is of crucial importance in its potential for making feasible the evaluation of a vastly larger set of engineering systems.

Two classes of simulation speed up methods have been examined in the literature: variance reduction methods (see Wilson (1984), Iglehart and Glynn (1987), Ripley (1987), Kleijnen (1974-75), Parekh and Walrand (1986)) and distributed simulation methods (Mishra (1985), Chandy (1988), Jefferson (1985), see Righter and Walrand (1988) for a literature survey).

The former class is based on the observation that the number of replications of a simulation experiment necessary to obtain an estimate with a prescribed level of confidence is proportional to the variance of the estimator. Thus, by constructing an estimator with a smaller variance, one can reduce the number of necessary experiments, and hopefully thereby speed up the simulation. In general, these variance reduction techniques are difficult to implement and they rarely produce a substantial variance reduction. This explains why very few commercial simulation packages exploit these methods.

Distributed simulations present a very promising potential. They are based the fact that a multiple processor machine can achieve a higher computational power at a more attractive price in terms of design cost, than a single processor design. For instance, the NCUBE computer with 512 processors has a potential processing power comparable to that of a CRAY-XMP, which is 15 times more expensive. It appears that the evolution of technology will continue to increase the performance/cost advantage of multiple processor architectures. See Gupta (1987) for a survey of multiprocessor designs. The challenge is however to design simulations that make good use of that potential power of multiple processor systems. For instance, if a simulation on a 512 processor hypercube uses each processor 50% of the time, then the simulation will be 256 times faster than on a single processor machine, and then will achieve a performance/cost about 7 times better than a CRAY X-MP.

Simulations of networks also tend to be exceedingly slow. For instance, consider a Jackson Network with 40 nodes, and with a traffic intensity of 50% at each node. The average number of transitions in a busy cycle can be estimated as follows. Denote the total arrival rate into the network (from outside) by λ. If the average busy cycle duration is B, then one must have $\alpha(B+\alpha)^{-1} = \pi(0)$ where α is the average holding time of the empty state and $\pi(0) = 2^{-40}$ is the stationary probability of that empty state. Now, $\alpha=\lambda^{-1}$, so that $B \approx \lambda^{-1}2^{40}$. As a consequence, the average number of customers served in a busy cycle is $\lambda B \approx 2^{40} \approx 10^{12}$. To determine the average time required to simulate a busy cycle of the network, one argues that each customer will require an average number of instructions of the order of 10 to 100. This rough estimate considers the average number of queues that a customer has to go through (this depends on routing probabilities) and the number of instructions required to process each arrival and departure at a queue (generating random service times, updating statistics collection routines, making routing decisions). This leads to an estimate of between 10^{13} to 10^{14} instructions per busy cycle. Thus, on a 10 MIPS machine, the simulation of a busy cycle will take between 11 days and 4 months. To be statistically significant, the simulation should generate a number of busy cycles if we want estimates about the stationary distribution.

Numerous examples have shown that there is no universally preferred simulation approach on a multiprocessor machine. Similarly, the characteristics of the multiple processor

This research was supported in part by the Computer Sciences Division, Shell Development Co., Houston, TX, and in part by the National Science Foundation (NSF).

machine affect the selection of a suitable task assignment algorithm. We will discuss these issues in later sections. A "vectored" simulation approach, Buckshot, will be discussed to outline some methods which may help improve a suboptimal initial assignment of tasks to the distributed computing system (see Madisetti, Walrand and Messerschmitt (1988b)). An objective of our research is to identify the relevant characteristics of a multiple processor machine and that of the system model and then to provide guidelines for the selection of the appropriate simulation methodology.

2. SIMULATION METHODOLOGIES

In a synchronous event-driven simulation, the processors have access to the "local time" of other processors, either through a centralized global clock, or through message passing. Thus, at each transition, a processor can tell whether its local time is the next one to be updated. Special hardware may be used to speed up this clock distribution.

In a conservative asynchronous event-driven simulation, the processes are updated only when all the relevant information is available. One method for achieving this is to have each process inform others of the earliest time it might send them some information. This would allow receiving nodes to schedule their event list with some freedom, and also keep updating their local times. This method also prevents deadlocks if each process adds a small positive increment of simulation time, between the input and the output timestamps. Another method is to accept the inevitability of deadlock, and send "probes" to elicit the required information. A few variations of these schemes have been proposed that permit deadlocks to occur, be detected, and then be resolved. See Mishra (1984).

By contrast to the preceding method, in an optimistic asynchronous event driven simulation, the processors "charge ahead" as fast as they can. As a consequence, it may be that some of the updates were performed erroneously (because all the relevant information was not available). When this occurs, the processes "roll back" their local clocks to the latest state compatible with the new information sending "anti-messages" to undo the erroneous messages which were sent with partial information. Variations of this time warp approach, involving the timing of these messages have been proposed (timewarp with aggressive or lazy cancellation). See Jefferson and Gafni, (1988).

Our discussion now focuses on developing models for the timewarp mechanism and on a new algorithm, Wolf, for optimistic simulation of dynamical discrete event systems. Wolf limits the loss of causality to a *sphere of influence* , $W_m(T)$, where m is the error message detected to be an error at time T after it had been processed (Section 3).

2.1 Time-Warp Mechanisms

In this section, we analyze two different rollback strategies proposed by Jefferson and Gafni (1988). Their mode of a execution is basically the same with an identical number of messages required to be corrected, however, some forward computation and communication costs maybe be saved in the case of lazy cancellation at the cost of some additional memory. Exactly when these advantages may be realized is a pertinent problem, and very often the solution lies in the properties of the messages in the flow. We now establish some terminology for timewarp developed by Jefferson, Gafni and others for purposes of analyses.

Each process in the network keeps a Virtual Time clock, denoted as LVT which is updated by each message the process receives. A process p receives event messages in its Input Queue and continue processing them until either no event messages remain or a message is received whose time stamp is lower than that of the local LVT. This message is appropriately termed the *straggler* and triggers the onset of the rollback in simulation time. The LVT then has to be rolled back to the timestamp of the straggler. To assist in the rollback process, all messages processed prior to the receipt of the straggler are kept in a queue termed the Output Queue. Each of these records is called an *antimessage* for the previously sent message. Therefore, rollback is implemented by sending antimessages to subsequent recipient processes. When an antimessage meets its corresponding message it *annihilates* it. Each process treats the antimessage *exactly* like it would treat any other message in the Input Queue, and process it according to a timestamp order. The antimessages are then propagated throughout the network until the rollback wave spreads to all the processes and the times for all processes are consistent with the causal assumption.

Of interest is the time for which the rollback wave propagates in the network. This represents the lost time which would otherwise have been productively utilized in marching forward in simulation time. Memory requirements are also a focus of interest as most commercially available have a limited memory per node. Some remedies for this problem will be outlined later in the paper. We will now briefly discuss implementations of standard timewarp: 1. Aggressive Cancellation, and 2. Lazy Cancellation

The rollback mechanism consists of two phases; (A) A Restoration Phase, and (B). A Cancellation Phase. In the Restoration Phase, the current state is destroyed and the Input Queue and the Output Queue have pointers set to the state with the highest simulated time strictly greater than the timestamp of the straggler V. The straggler is enqueued in the input queue and is ultimately received by the process p, which then sets its LVT to V. In the Cancellation Phase, all the states in the State Queue of the process p with time stamps greater than V are erased, and all antimessages corresponding to messages with Send Times (ST) greater than V are sent to the receiving process r. There, the antimessages are enqueued and annihilate their corresponding messages to rollback to a state consistent with the available information. The receiving process r , then generates successive antimessages for its recipient processes. This mechanism is termed as **Aggressive Cancellation,** since it cancels all the messages with time stamps greater than V.

The Restoration Phase for **Lazy Cancellation** is the same as for Aggressive Cancellation, however, the Cancellation Phase is different. The process p , does not send the antimessages immediately, but sends antimessages for those messages which were found altered on recomputation. It is anticipated that only a few messages need be retransmitted resulting in a reduced number of cancelled messages. Two parameters suggested for evaluating the efficacy of rollback schemes have been the number of rollbacks generated by a single straggler communicated between two processes, and the number of antimessages transmitted (equal to the number of messages cancelled). The operation of standard time-warp algorithms and the proposed Wolf algorithm will be compared using an example, as illustrated in the following paragraph.

2.2 A Rollback Example

Let the straggler arrive at process p with a time stamp V. The process p is at the some time stamp higher than V, say H. Therefore, p has to rollback from H to V. Let OUT be the first event in the Output Queue of p with a time stamp greater than H. This corresponds to the antimessage for the last message sent by p. Let NEW be the new message generated by p to recompute the true value based on the straggler's information. NEW is then transmitted by p to the receiving process r. We observe the cancellation phase in process r as a result of these messages. In real time t_0 is the time when the straggler is received by process p, t_1 is the time when message *OLD* is received by r and t_2 is the time when r receives the message *NEW*. We only consider the first stage of rollback before it propagates to other receiving processes (Section 3) Let us also consider the case when ST(NEW) < ST(OLD). See also Gafni and Jefferson.

Aggressive Cancellation: The message OLD arrives at time t_1 to process r, let X_2 be the number of messages in r's Output Queue with time stamps > RT(OLD), and r computes forward Y messages, and then, when finally NEW arrives at real time t_2 there are X_1 messages with time stamps between RT(NEW) and RT(OLD). The total number of messages to be cancelled is X_1+X_2+Y messages. Figure 1 shows this graphically.

Lazy Cancellation: Lazy cancellation has to cancel the same number of messages, however, it saves on the transmission of identical messages. There is a factor μ which determines the efficiency of a lazy evaluation.

In the next few sections, we examine rollback in a general framework. Some embedded source models, and the notion of spheres of influence are introduced for the purpose of analysis of optimistic computing systems. A new rollback algorithm Wolf, is then examined in this framework.

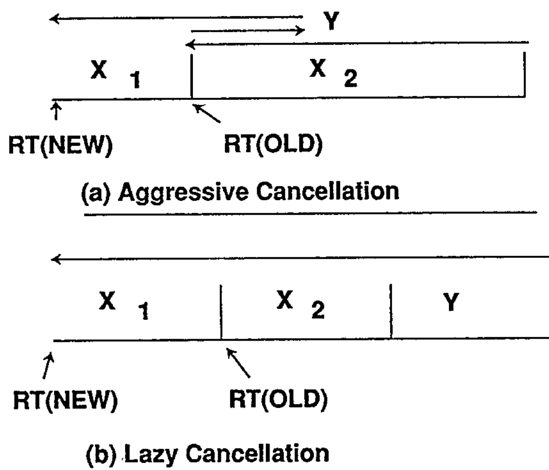


Figure 1

3. A ROLLBACK ALGORITHM: WOLF

In this section, we make precise the underlying structure and mechanisms through which elements (processes) in a dynamic discrete event system interact with each other. The efficiency of a distributed implementation of a simulation is dependent on a number of factors: 1) The concurrency inherent in the system being simulated, 2) The potential parallelism that can be extracted, and, 3) The communications overhead in passing data and control variables within the system. Very often, as in most optimistic simulation algorithms, the structure a the system being simulated is not utilized in planning the simulation. To aid such an implementation we classify DEDS into 1) Static Systems, and 2) Dynamic Systems. These classes will be made precise in the next few paragraphs. Static systems allow efficient implementation of the algorithms proposed in this paper, dynamic systems may need more memory and a relatively higher communication bandwidth. The class of static systems is, however, sufficiently rich to include a number of interesting systems.

3.1 Network Models

Static Systems:

Consider a network of computing nodes, 1 , ... , N. Let i and j be any two nodes in that network. If r_{ij}^{cd} equals one then we assume node i sends a message m of class c to node j where it belongs to the class d. Here $c, d \in$ C and $i, j \in \{1, 2, 3, 4, ., , N\}$. If

$$Prob\ [r_{ij}^{cd} = 1] = p_{ij}^{cd} \quad \text{for all } i, j, c, d$$

and

$$\sum_j p_{ij}^{cd} = 1 \quad \text{for all } i, c, d$$

Then the system is defined to be a **static** system. Static systems allow the modeling of a variety of communication networks, manufacturing systems, distributed computing systems and transportation systems amongst others. In essence, the message follows one of several different paths and there is no explicit dependence of the routing on the states of the transmitting and the receiving nodes.

Dynamic Systems:

If the probability of transmission of the message m from node i to node j depends explicitly on m and the states of the nodes, then the system is defined as a **dynamic.** In such cases, a logical network itself does not exist and the system is a collection of interacting particles. Examples of such systems include battlefield simulations, interactions between atomic particles among others.

$$Prob\ [r_{ij} = 1] = p^m{}_{S(i),S(j)} \quad \text{for all } i, j, m$$

Where $S(i)$ and $S(j)$ denote the states of nodes i and j.

3.2 Sphere of Influence, $W(i, t)$

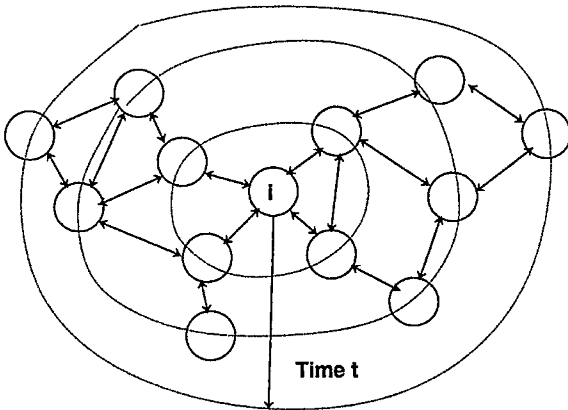
In this section, we wish to quantify the notion of error propagation within an optimistic distributed simulation system. Let b be the communication time between two neighboring nodes, and a be the processing time required to process a single message by any computing node. If a message of class c completes

its processing at time t=0 in node i, then we define the set $W(i,t)$ to be the set of nodes that can be influenced by that message in time t. More specifically,

$$j \in W(i,t) \Longleftrightarrow p_{kj}^{\alpha\beta} > 0 \text{, for some } \alpha, \beta \text{, and}$$

$$k \in W(i, t-a-b)$$

As the communication times b and computation times a are in general random, replacing them by the minimum communication times and processing times leads to a conservative estimate of the sphere of influence. A finite algorithm allows one to compute $W(i,t)$ for any finite t. This can be done off-line. This information can be stored in node i, possibly after some compression by approximation in the form of a lookup table. The sphere of influence could also be adaptively updated to monitor the changes in the network. The radius of propagation, $R(i,t)$, of the sphere of influence, $W(i,t)$, is the distance in the number of nodes, which a message transmitted by node i could propagate in the time t. For a conservative estimate, the radius of propagation would be $\frac{t}{a+b}$. The sphere of influence (in the conservative sense), can be looked upon as consisting of a number of shells of increasing radii $(1-R)$, each shell consisting of nodes reachable from i within a certain time span. The sphere of influence thus enables the simulation designer to take advantage of the structure the system being simulated. For example, even if a certain computing node could communicate (directly and indirectly) with another, the requirements of the simulation could preclude such communications, and this knowledge can be used to prune the sphere of influence [Figure 2].



Sphere of Influence, W (i , t)

Figure 2

3.3 Wolf Algorithm for Rollback

Notation:

OQ_k is the Output Queue of Node k.
IQ_k is the Input Queue of Node k
S, is a message which is the Straggler.
LVT is the Local Virtual Time at a node.
E is the Error Message detected at time t_S after it was processed.
$A+$ denotes the messages which were processed after A was processed.
$i \Rightarrow j \in W \mid A$, implies that i broadcasts message A to nodes $j \in W(i,t_S)$
S_R consists of those nodes, $k \in W(i,t_S)$, which are at a radial distance of R away.

The following is the algorithm for the node, i, which initiates Wolf to rollback the effects of error message, E, processed at real time of t_S seconds earlier.

Wolf Algorithm

Node i:

$i \Rightarrow j \in W(i,t_S) \mid V(E,T_E)$
Rollback LVT to T_i
Await ACK_j from all $j \in S_R$
Initiate Forward Compute Phase
End

/* $V(E,T_i)$, is the Wolf-call, containing the identity of the error message, E, and the timestamp, T_E, at which it was received by i. */

/* Each receiving node, j, processes $V(E,T_E)$. We illustrate the algorithm for node $j \in W(i,t_S)$ when it receives a Wolf-call, $V(A,T)$. */

Node j:

Read $V(A,T)$.
While (LVT_j < T) *do* continue processing *enddo;*

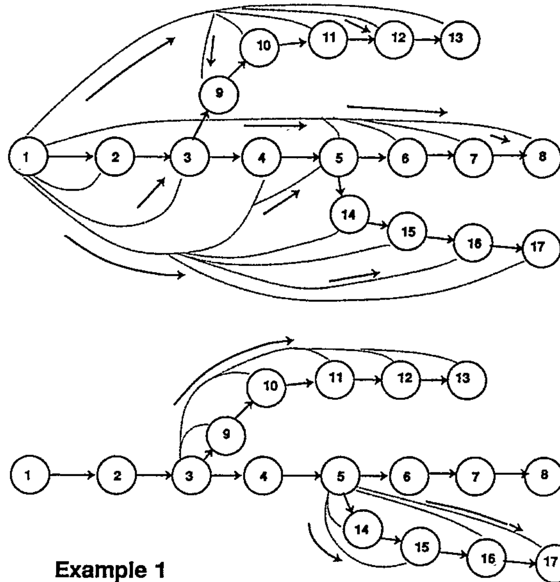
If ($A \cap OQ_j \neq \phi$) *then* $j \Rightarrow k \in W(i,t_S) \mid V(\Omega+, T_{\Omega+})$

/* Here $A \cap OQ_j = \Omega$.*/
/* Here T_A denotes the LVT when j received message A. */

Rollback to T_A
If $j \in S_R$ transmit ACK_j to i
else await $V(A,T)$
endif
else await $V(A,T)$
endif

The algorithm ensures that the effects of the error message, E, are limited to the sphere of radius R, $W(i,t_S)$. In addition, only P broadcasts are necessary to complete the rollback as opposed to at least R communication steps required by timewarp. P is usually much smaller than R. In the next section, we outline how P can be determined.

We need next to estimate the communication overhead for Wolf. If the time required for a single broadcast is ε then for P broadcasts we need time $P\varepsilon$. At each step of the algorithm, a few nodes (on an error path) broadcast Wolf-calls concurrently, incurring overhead of only one broadcast. Therefore P, the number of steps the algorithm takes to terminate, determines the overhead in communications. In practice, the time required for a multihop broadcast is approximately equal to that of a single hop communication step. This performance is achieved using a virtual cut-through routing algorithm.



Example 1

Figure 3

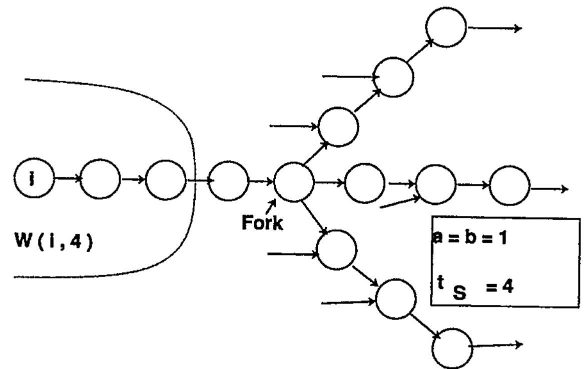
Let us now consider a few examples, to illustrate rollback using Wolf. In Example 1 [Figure 3], we have a simple queueing network with a sphere of influence, $W(i,t_S)$ consisting of the 17 nodes as shown. Node 1 initiates Wolf when it detects the straggler. Let us assume that the error message, E, is now at node 8. Therefore, nodes 1-8 form the *primary* error path, or the error path of *order 1* as we will call it. When node 1, initiates the rollback, it broadcasts information about the error message and the its Local Virtual Time when it received the error message, E. Each receiving node stops processing its Input Queue if its LVT is greater than that time, and in addition, if it has actually processed E then it is on the primary error path and rolls back (to the LVT at which it received E). It then broadcasts to the remaining nodes (off the primary path) in $W(i,t_S)$, information regarding the messages it has processed subsequent to processing E. In this case, nodes nodes 3 and 5 are the only nodes on the primary error path (order 1) which have transmitted messages along paths 3-13 and 5-17 (order 2). These broadcasts take place simultaneously, with all the nodes rolling back to the LVTs which are consistent with the available information. In short, the primary path 1-8 rolls back after the first broadcast, and then the paths of order 2, 3-13 and 5-17, roll back after the second broadcast.

If standard timewarp (without broadcast and "blocking") were used, a minimum of 7 communication hops would be required, as the antimessage is propagated along the nodes 1-8 in succession and likewise along the paths 3-13 and 5-17. If the the nodes were lightly loaded then the number of communication steps required by the antimessage to meet with and *annihilate* the error message would be much larger. Forward computation is then initiated by Node 1, and the nodes restart forward computation. Forward computation after rollback can use with with advantage the facts that lazy cancellation can be used (to save on retransmission of messages) and that random numbers previously generated at the nodes can be reused to simulate service times.

We have introduced some new notation in the previous example; that of the *order* of the error path. Paths of order 1, contain those nodes which had processed the primary error message E. Paths of higher order are those which originate from the paths of lower order, when nodes on lower order paths process messages subsequent to processing an error message (and hence those messages are also errors and need be corrected). This notion allows us to determine the number of broadcasts which would be required by the Wolf algorithm.

The number of broadcasts required by Wolf equals the highest order of any error path in the sphere of influence.

In the case of Example 1, the highest order of error paths was 2, hence 2 broadcasts were required. It can be easily shown that the maximum number of broadcasts would be limited by the radius of propagation of the sphere of influence. In Example 1, this radius was 7.



Example 2

Figure 4

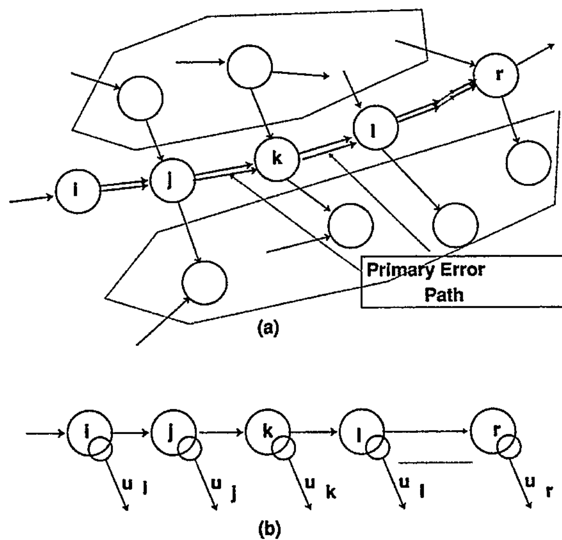
Our next example [Figure 4], depicts the configuration of a manufacturing system. For the purposes of illustrating the penalties of not using broadcast algorithms for rollback, we assign values for the minimum communication time between

nodes, $b=1$, seconds and the minimum processing time , $a=1$ seconds. If the error was detected by node 1 at time t=4 seconds, and the network were lightly loaded, it is unlikely that the antimessage transmitted by node 1, would meet with the error message to annihilate it. If the error message reached the fork, secondary error messages would contaminate the entire network. Wolf, however, with a combination of broadcast and "blocking" guarantees that effects of the error message will be confined to the sphere of influence (with radius 3). Besides, all the nodes in the error path roll back in simulated time concurrently. Since, only a few paths span the sphere of influence the rollback phase with Wolf is very short, implying a short recovery period from the loss of causality.

Based on this discussion, we formalize the notion of primary and secondary error paths. We consider two paths differing in order by one, the one with the lower order being the primary path and the one with the higher order, the secondary error path.

3.4 Embedded-Source Model for Rollback

Let node i process a message E at real time t=0, and at time $t=t_S$ a *straggler* is detected, informing node i that message E was an error message. Message E, in the mean while, has been processed by nodes j , k , $l \cdot r$ when the straggler arrived at i. This path $i-j-k-l \cdot r$ is defined as an error path of **order 1.** Each node along the error path of order 1, processes other messages subsequent to processing the error message, these messages are also error messages and directly influence the rest of the network.



Embedded-Source Model for Analysis

Figure 5

In a dynamical discrete event system, with random routing, the path of order 1, gives rise to a number of secondary error paths of order 2. For the purpose of analysis [Figure 5], each of the nodes, n, on the path of order 1 is embedded with a source of rate μ_n (messages/sec), which generates error messages to the rest of the network. The messages from i are processed as they arrive. Therefore, between every two error messages processed along the primary error path, a number of secondary error messages resulting from interaction with the rest of the network are generated.

In the path of order 1, nodes j , k , l ,...,r are at radii of 1 , 2 , 3 ...., R respectively. The number of error messages generated by these R nodes in the time t_S would be

$$(t_S - b - a)\mu_j + (t_S - 2b - 2a)\mu_k + (t_S - 3a - 3b)\mu_l + \ldots + (t_S - Ra - Rb)\mu_r$$

These messages represent those which are generated prior to the time the straggler is detected. When Wolf is invoked, additional error messages are generated in the time it takes for the broadcast to reach the nodes, and their number is

$$\varepsilon(\mu_j + \mu_k + \mu_l + \cdots + \mu_r)$$

However, if there were no Wolf-call and antimessages are propagated from node to node, then number of additional error messages that remain to be cancelled are given by

$$(b+a)(\mu_j + 2\mu_k + 3\mu_l + \cdots + R\mu_r)$$

The number of additional error messages to be cancelled depends on the rates of the embedded sources, and is particularly sensitive to the sources on nodes at increasing radii of propagation (owing to a linear multiplier). Therefore, a long error path is capable of generating a large number of secondary error messages relative to a short error path (small t_S). Wolf, however, is insensitive to the path length. In this simple example, we have considered two paths; the analysis can be extended to paths of higher order. If all nodes in the sphere of influence were not informed about the straggler then the numbers given above represent only the lower bound on the additional messages that need be cancelled.

3.5 Pipelined Forward Computation and Rollback

Spheres of influence allow an elegant representation of forward simulation and rollback in an optimistic distributed simulation. The sphere of influence of an error message grows with time, until the error is detected and Wolf is invoked. The the broadcast Wolf-call "freezes" the sphere of influence while the rollback phase begins.

In Wolf, nodes in more than one shell can rollback concurrently. The signal to rollback does not depend on the radius of the shell a particular node is in, but on the order of the error path of which it is part. This is unlike standard timewarp, where the shells rollback in succession, starting with the shell at radius 1 and ending with the rollback of shell at radius R. In Wolf, we have P broadcasts and hence P phases in the rollback. However, both the schemes allow pipelining of forward computation with rollback. In Wolf, a phase could include rollback of more than one complete shell. We must note, however, that restarted forward computation though faster is still sequential. Feedback paths may slow down forward computation.

After the nodes in a shell have rolled back, they restart forward computation, while the nodes in the shells of greater radii begin to roll back in simulated time. This pipelining then progresses until all the shells in the sphere of influence have rolled back to a state consistent with the available information. By then, the entire sphere will have restarted forward computa-

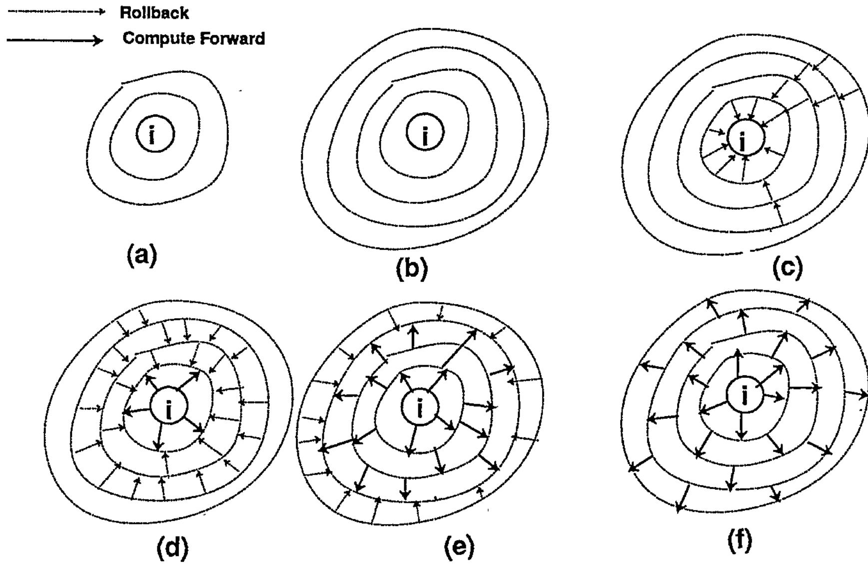


Figure 6

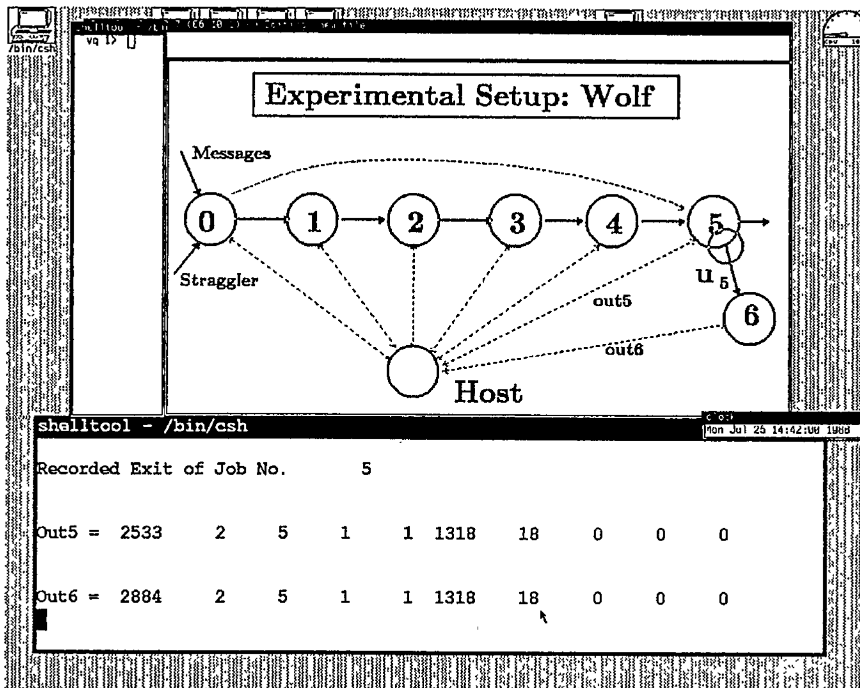


Figure 7

tion. This sequence is shown in Figures 6(a)-(f). In (a) the sphere of influence is growing to that in (b). At the instant (b), the node i detects the straggler, and the rollback starts as shown by (c). In (d), shells 2 and 3 rollback, while shell (1) computes forward. In (e), shell 4 rolls back, while shells 1, 2 and 3 compute forward. The Wolf phase is completed in (f).

4. DESIGN OF SIMULATORS

In this section, we delve briefly on some strategies to plan and implement large scale simulations of dynamical discrete event systems on a network of computing processors. Such systems contain a large number of computing nodes (40-50), flexible routing facilities, service disciplines, and monitoring and statistics collection routines.

The initial phase of the simulation involves formulating a model for the real life system being simulated. The onus of designing a precise model of the dynamical discrete event system rests largely on the application expert. The next step is to distribute the model onto a set of logical processes. Then the processes are mapped onto a distributed computing system with a number of computing elements. Since the concurrent computing environment consists of a number of concurrent and, in general, asynchronous processes sharing a few common processors, an efficient task assignment and load balancing algorithm must be used to schedule the distributed simulation. We have developed a Concurrent System Programming Language (COSPROL), to adaptively assign computing resources in a distributed multitasking, multiprogramming environment (See Madisetti and Messerschmitt 1988b). Both conservative and optimistic distributed simulation methods can improve upon the efficiency of their implementation by minimizing the communication overhead. The overhead in conservative methodologies arises from the null messages that are communicated between processes to maintain causality. In optimistic schemes, communications are necessary to rollback to a state consistent with the available information. In another paper (See Madisetti, Walrand and Messerschmitt, 1988a, and Heidelberger, 1986, for a specific case), we have proposed an algorithm through which a number of messages from B independent simulations are processed on the same distributed simulation testbed. Instead of a single message, a "vector" consisting of B messages from B independent simulations are transmitted between processes. All the messages in a "vector" share the same communication overhead and also help keep processors busy. For example, the null messages of one simulation are sent with the real jobs of another orthogonal simulation run. In the case of optimistic Wolf, the processors can compute forward in one simulation run, while they are blocked by rollback in another independent simulation run. The two main advantages are the reduction in the communication overhead, and the increase in the utilization of the computing elements. We define this approach as "vectored simulation" and call B as the Buckshot vector.

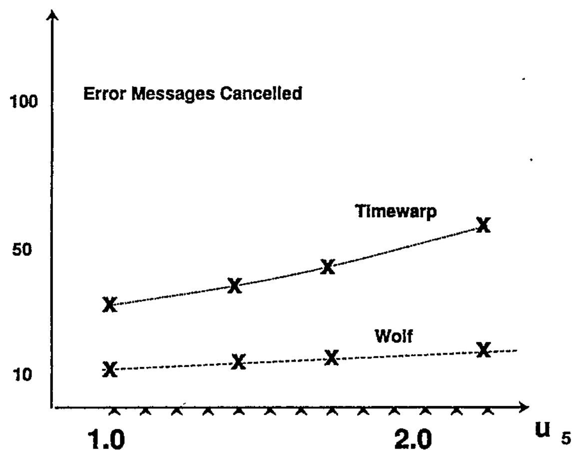
Once a suitable "vector" length is chosen, the next step is the identification of Wolf nodes and determining their spheres of influence. Both of these depend on the structure of the distributed model being simulated.

Wolf nodes can be usually identified as those with more than one input. The sphere of influence can be determined by a number of methods. In the algorithms discussed in this paper, we have assumed that the sphere of influence is known, and the efficiency of the algorithm depends on how well this sphere can be estimated in practice (hence the name, Wolf). A conservative method, in which the minimum communication and processing times are used is described in the paper. This may overestimate the radius of the sphere, especially when the network is heavily loaded. Other approaches we are considering, include a dynamic algorithm which iteratively evaluates smaller spheres of influence and extrapolates to a large one. For example, to evaluate $W(i,T)$, at first $W(i,T_1)$ is determined for $T_1<T$, and then the nodes, k in $W(i,T_1)$ determine $W(k,T-T_1)$. The union of these spheres determines the required sphere of influence. This approach can be extended to evaluate the sphere of influence as accurately as possible. However, occasional errors can result due to changes in the states of the queues, in dynamic systems. Another method, would be to release marked messages occasionally and determine their trajectories with time. Very often, especially in static systems, the entire trajectory of a message through a system can be probabilistically determined before the message enters the network, simplying evaluation of $W(i,t)$.

5. EXPERIMENTAL RESULTS

In this section, we present some results from some pilot implementations of rollback algorithms on the NCUBE distributed memory multiple processing system. The NCUBE multiple processor system, connects upto 1024 computing elements each with its own local memory. The processors communicate via asynchronous message passing. Each of the nodes is capable of about 0.5 MFLOPS peak performance and a strong hypercube type interconnectivity allows concurrency in communications. Latest models of commercial multiprocessors provide very efficient multi-hop communication protocols (See Dally, 1987), with the time taken for a multi-hop message communication step only slightly larger than that of a single hop. In addition hardware broadcast facilities are provided, enabling a single node to broadcast to a number of other neighboring nodes through dedicated links.

Wolf and timewarp algorithms were implemented on a network of seven computing nodes. The network topology is illustrated in Figure 7 . The objective of the experiment was to experimentally verify the embedded-source model and the sensitivity of the messages to be cancelled to the source rates and the path length. Extensive simulation of Wolf, on a large network using about 40 nodes is currently being implemented to study the performance tradeoffs in practice. In this example, the nodes were strung in tandem, and node 5, is embedded with a source of rate μ_5 modeling interaction with the rest of the network. The performance of Wolf and timewarp was then measured with varying processing times. One such set of observations is illustrated in Graph 1. The numbers represent the number of error messages that need to be cancelled by either scheme. The straggler was separated from the error message by 6 time units. When two embedded sources were used (the other on node 4), the numbers of error messages were an order of magnitude higher. We will present an exhaustive comparison in a later paper, when a realistic dynamical system will be simulated. Our preliminary results are encouraging, indicating an order of magnitude reduction in the number of messages to be cancelled by Wolf relative to timewarp, when two paths are considered. (the sum $(a+b)$ along the path is normalized to 1 and the source rate is varied between 1 and 2).



Graph 1

6. CONCLUSIONS AND FUTURE WORK

We have presented a new analytical model for optimistic distributed simulation systems. A new algorithm for rollback, Wolf, is proposed and its performance discussed. Wolf has the advantages of limiting the error propagation to the sphere of influence, and a fast recovery from errors. A few preliminary implementations of Wolf, on a commercial multiple processor system, confirm predicted results. Detailed simulation experiments are being planned to determine the performance of Wolf in the optimistic simulation of large scale dynamical discrete event system.

7. ACKNOWLEDGEMENT

The authors acknowledge gratefully the careful review by Dr. John Gilmer, BDM Corporation, Va, and by Dr. Teresa Meng of Stanford University. Comments from Professor Robert Brodersen are also acknowledged.

8. SELECTED REFERENCES

Berry, O. and Jefferson, D. [1985], "Critical path analysis of distributed simulations," Proc. SCS Dist. Simul. Conf., 57-60.

Chandy, M. [1988], "Distributed simulation tutorial," *SCS Multiconf. on Distributed Simulation.*

Chandy, K.M. and Reynolds, P.F. [1977], "Scheduling partially ordered tasks with probabilistic execution times," *Proc. 5th Symposium on Operating Systems, A.C.M.,* 167-172.

Cohen, G., Dubois, D., Quadrat, J. P., and Viot, M. [1985], "A linear-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing," *IEEE Trans. Aut. Control,* **AC-30**, 210-220.

Dally, W. [1987], "Wire-Efficient VLSI Multiprocessor Communication Networks," *Proc. of the Stanford Conference on VLSI, 1987,*, 390-415.

Fayolle, G., King, P., and Mitrani, I. [1983], "On the execution of programs by many processors," *PERFORMANCE '83, , 217-228.*

Fujimoto, R. [1987], "Performance Measurements of Distributed Simulation Strategies," *Techincal Report UUCS-87-026a, Univ. of Utah.*

Gafni, A. [1988], "Rollback Mechanisms for Optimistic Distributed Simulation Systems," *Proc. of Distributed Simulation Conf., SCS, San Diego. '88.*

Glynn, P. and Iglehart, D. [1987], "Importance sampling for stochastic simulations," *Tech. Report No. 49, Dept. of O. R., Stanford University.*

Gupta, A. (Ed.) [1987], "Multi-Microprocessors." *IEEE Press.*

Heidelberger, P. [1986], "Statistical analysis of parallel simulations," Proc. Winter Simulation Conference.

Ho. Y. C. [1985], "A survey of the perturbation analysis of discrete event dynamical systems," *Ann. Oper. Res.* **3**, 393-402.

Hu, T.C. [1961], "Parallel sequencing and assembly line problems," *Oper. Res.,* **9**, 841-848.

Kleijnen, J.P.C. [1974-5], "*Statistical Techniques in Simulation, Part I and II*". Marcel Dekker.

Klimov, G.P. [1974], "Time sharing service systems I," *Th. Prob. Appl.,* **19**, 532-551.

Mishra, J. [1985], "Distributed Discrete-Event Simulation",. *Computing Surveys,* November 1985

Livny, M. [1985], "A study of parallelism in distributed simulation," *Proc. SCS Dist. Simul. Conf.* 94-98.

Madisetti, V ., Walrand, J., and Messerschmitt, D. [1988a], "A High Performance Simulation Methodology for Dynamical Discrete Event Systems." *Proc. of SCS Western Multiconference on Simulation and Modeling, Jan 2-4, 1989, San Diego.*

Madisetti, V., Messerschmitt, D. [1988b], "Distributed computation on Concurrent Processors." *Submitted for publication.*

Marsan, M.A., Balbo, G., and Conte, G. [1986], "Performance Models of Multiprocessor Systems." *M.I.T. Press.*

Nelson, R., Towsley, D., and Tantawi, A. N. [1987], "Performance analysis of parallel processing systems." *to appear in IEEE Trans. Soft. Eng.*

Parekh, S. and Walrand, J. [1986], "Quick simulation method for excessive backlogs in networks of queues." *IEEE Trans. Aut. Control,* to appear.

Righter, R. and Walrand, J. [1988], "Distributed simulation of discrete-event systems." *Submitted to Proceedings of IEEE .*

Ripley, B. D. [1987]. *Stochastic Simulation* J. Wiley.

Walrand, J. [1988], *An Introduction to Queueing Networks,* Prentice-Hall.

Jefferson, D. [1985], "Virtual Time", *ACM Trans. on Programming Languages & Systems,* July 1985.

AUTHORS' BIOGRAPHIES

Vijay Madisetti graduated from the Indian Institute of Technology, Kharagpur, India, in 1984, with a B. Tech (hons) degree in Electronics and Electrical Communications Engineering. Since then he has been with the department of Electrical Engineering and Computer Sciences at the University of California at Berkeley, as a candidate for the Ph.D degree in Electrical Engineering and as a Asst. System Specialist with the Electronics Research Laboratory (ERL), Berkeley. His interests are in communication networks, signal processing and algorithms for distributed simulation and parallel computation.

Jean Walrand is a native of Belgium. He got his Ing'enieur Civil degree in Electrical Engineering from the Universit'e de Li'ege, Belgium in 1974 and his Ph.D. in Electrical Engineering from the University of California at Berkeley in 1979.

Dr. Walrand served as Assistant Professor with the School of Electrical Engineering of Cornell University from 1979 until 1981 when he joined the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley where is now Associate Professor.

Dr. Walrand served as Associate Editor of *IEEE Transactions on Automatic Control* and is presently Associate Editor of *Probability in the Engineering and Informational Sciences, Queueing Systems: Theory and Applications,* and *Systems and Control Letters.* He is the author of *An Introduction to Queueing Networks,* (Prentice-Hall, 1987).

His research interests are in Communication Networks, Queueing Systems, Stochastic Control, and Stochastic Processes.

David G. Messerschmitt is a Professor of Electrical Engineering and Computer Sciences at the University of California at Berkeley. From 1968 to 1977 he was a Member of Technical Staff and later Supervisor at Bell Laboratories, Holmdel N.J., where he did systems engineering, development, and research on digital transmission and digital signal processing (particularly relating to speech processing). Current research interests include applications of digital signal processing, adaptive filtering, digital communications (on the subscriber loop and fiber optics), architecture and software approaches to programmable and dedicated hardware digital signal processing, communication network simulation, design and protocols, and computer aided design of communications and signal processing systems. He has published over 100 papers and has 10 patents issued or pending in these fields. Since 1977 he has also served as a consultant to a number of companies.

He received a B.S. degree from the University of Colorado in 1967, and an M.S. and Ph.D. from the University of Michigan in 1968 and 1971 respectively. In addition to being a Fellow of the IEEE, he is a member of Eta Kappa Nu, Tau Beta Pi, Sigma Xi, and has several best paper awards. He has served as a Senior Editor of the *IEEE Communications Magazine,* as Editor for Transmission of the *IEEE Transactions on Communications,* and as a member of the Board of Governors of the IEEE Communications Society.

The authors may be contacted at
Department of Electrical Engineering
and Computer Sciences,
University of California at Berkeley
Berkeley, CA 94720