Software Supports for Event Preemptive Rollback in Optimistic Parallel Simulation on Myrinet Clusters*

Andrea Santoro and Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy

Abstract

Optimistic synchronization protocols for parallel discrete event simulation employ rollback techniques to ensure causally consistent execution of simulation events. Although *event preemptive rollback* (i.e. rollback based on timely event execution interruption upon the arrival of a message revealing a causality inconsistency) is recognized as an approach for increasing the performance and tackling run-time anomalies of this type of synchronization, the lack of adequate functionalities at the level of general purpose communication layers typically prevents any effective implementation of event preemptive rollback operations.

In this paper we present the design and implementation of a communication layer for Myrinet based clusters, which efficiently supports event preemptive rollback operations. Beyond standard low latency message delivery functionalities, this layer also embeds functionalities for allowing the overlying simulation application to efficiently track whether a message will actually produce causality inconsistency of the currently executed simulation event upon its receipt at the application level. Exploiting these functionalities, awareness of the inconsistency precedes the message receipt at the application level, thus allowing timely event execution interruption for activating rollback procedures.

We also report experimental results demonstrating the effectiveness of our solution for a Personal Communication System (PCS) simulation application. The results show that with our implementation we get an improvement of up to 13% in the execution speed of the parallel simulation application, compared to standard, non-preemptive rollback operations, thus achieving speedup on the order of 60-80% of the ideal one on a Myrinet cluster of 8 LINUX machines.

Keywords: Parallel discrete event simulation, Optimistic synchronization, Rollback-recovery, Event preemption, Performance optimization.

*This article appeared in Journal of Interconnection Networks, Vol.6, No.4, December 2005. An earlier version of this article by the same authors appeared in the Proceedings of the 7th IEEE Symposium on Computers and Communications (ISCC'02).

1 Introduction

In parallel discrete event simulation, distinct parts of the system to be simulated are modeled by distinct Simulation Objects (SOs), each of which is essentially a sequential discrete event simulator having its own simulation clock, its own event list and its own state variables [7]. SOs are concurrently executed on a multiprocessor system or a cluster of machines, with the aim at reducing the completion time for the simulation application.

Any SO executes a sequence of simulation events and each event execution possibly schedules new events to be executed by any SO at later simulation time. The notification of scheduled events among distinct SOs takes place through the exchange of messages carrying the content and the occurrence time, namely the *timestamp*, of the event. In order to ensure correct simulation results, synchronization mechanisms are used to maintain a non-decreasing timestamp order for the execution of events at each SO.

Optimistic synchronization [10] allows each SO to execute events unless its event list is empty, and uses checkpoint-based rollback to recover from timestamp order violations, namely *causality inconsistencies*. A rollback recovers the state of the SO to its value immediately prior the violation. While rolling back, the SO undoes the effects of the events scheduled during the rolled back portion of the simulation and involving other SOs. This is done by sending an *antimessage* for each event sent out during the rolled back portion of the simulation. Upon the receipt of an antimessage that cancels an event already executed, the recipient SO rolls back as well. This is also referred to as "cascading rollback".

In current simulation software technology, event execution at a SO is handled as an atomic action, and any check for incoming messages/antimessages, which might reveal causality inconsistency, is performed only in between consecutive event executions. This means that rollback operations for a SO, if required, do never preempt any in-progress event execution activity [7]. Such a preemption exhibits, however, the following advantages:

(i) The simulation application does not waste CPU time to complete the execution of a causally inconsistent event that must be rolled back.

(ii) Interruption of the event execution might avoid sending some event notification messages to other SOs. Those messages, if sent, should be eventually revoked through the corresponding antimessages. Therefore, avoiding sending them might achieve a reduction of the communication overhead. Also, a reduction of the amount of messages to be revoked might reduce the likelihood of cascading rollback.

In addition, avoiding the send, and more in general the scheduling, of events that must be eventually revoked reduces the cost of managing event lists. Specifically, the event to be revoked is not delivered to any SO, therefore we avoid executing the operations required to insert that event into the appropriate event list.

(iii) Interruption of the event execution allows anticipating the sending of antimessages required by rollback operations. Therefore we get a reduction of the likelihood that, upon the antimessage receipt, the corresponding message was already processed by the recipient LP. This additionally contributes to reduce the likelihood of cascading rollback.

(iv) Beyond the previous performance issues, preempting event execution has the advantage of coping with some pathological situations of optimistic synchronization. Specifically, out of order processing of simulation events can lead some predicates to be verified, which would never be verified in a timestamp ordered execution. This might cause event processing routines to go into an infinite loop [17]. With event execution preemption, the system can recover from this situation.

The main reason why current simulation software technology does not employ event preemptive rollback is that general purpose communication layers are not content aware. Therefore, the only chance a SO has to become aware, during event execution, of any message/antimessage revealing a causality inconsistency, is to run the event routine and the message receipt module in time inter-leaved mode. However, this solution shows several drawbacks making it not viable. As a major drawback, event execution time can be stretched thus causing delay in sending notification messages for newly scheduled events. As shown in [3, 19], this possibly increases the likelihood of timestamp order violations at the recipient SOs, which might even give rise to rollback thrashing in the parallel execution.

In this paper we present software supports for event preemptive rollback in case of optimistic parallel discrete event simulation on Myrinet clusters. These supports allow the implementation of event preemptive rollback operations with no inter-leaving between event routine and message receipt at the application level.

Actually, Myrinet network cards [12], as well as other types of network cards of the last generation [9], are equipped with a programmable processor, typically used only for message transfer purposes. We employ that same processor to perform "on-the-fly" verification on incoming messages/antimessages, so as to ascertain whether the SO being executed is causally consistent. Then, the simulation application software reads the verification outcome at negligible cost by means of calls to a light function belonging to the API associated with the communication layer.

We test the effectiveness of these software supports by means of an experimental study on a Personal Communication System (PCS) simulation application. The data show that, with event preemption, rollback costs can be definitely reduced, with consequent increase (up to 13%) of the speed of the parallel execution.

The remainder of this paper is organized as follows. In Section 2 we cast a deeper look on event preemptive rollback and on the difficulties related to its implementation. In Section 3 we describe the software supports for event preemptive rollback we have developed. Performance results are reported in Section 4.

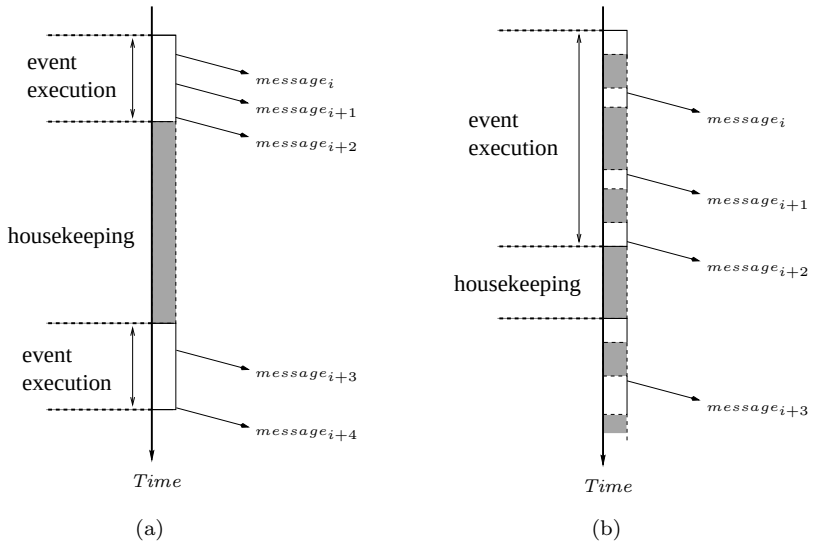

Figure 1: (a) Classical Rotation between Event Execution and Housekeeping. (b) Interleaving for Event Preemptive Rollback when Employing a General Purpose Communication Layer.

2 Background and Motivations

SO involved in optimistic simulation systems are characterized by the following two computation phases:

1. **Event Execution.** This phase involves:

   (i) Performing computation associated with a specific simulation event occurring at that SO, which possibly modifies the SO state (i.e. the state of the simulation model).

   (ii) Sending to other SOs notification messages for new events possibly scheduled by the current event execution.

   (iii) updating the SO event list with new events possibly scheduled by the current event execution.

2. **Housekeeping.** During this phase, several additional tasks are performed by the SO as a support to the parallel execution (e.g. receipt of incoming messages/antimessages and event list management, checkpointing-rollback, and so on).

Usually both phases are strictly atomic and, as shown in Figure 1.a, they exhibit an alternate execution. In other words, no housekeeping task is performed while event execution is carried out, and vice versa.

Event preemptive rollback, i.e. timely event execution interruption in case of causality inconsistency of the event currently being executed by the SO, is inviable in practice if general

purpose communication layers are used. With one of such layers, the simulation progress at a SO should exhibit the behavior shown in Figure 1.b. Specifically, housekeeping operations (i.e. receive operations) to determine whether incoming messages/antimessages reveal a causality inconsistency should be inter-leaved with the event execution itself. Inter-leaving is required since the layer is not aware of the content of transmitted/received information, therefore the check on causality inconsistency must be necessarily performed at the simulation application level, i.e. by the SO after the receipt of the information. This requires the receipt of messages/antimessages to be performed periodically while event execution is in-progress, with consequent data structures update (e.g. event list update) to be performed at the application level.

As shown by the picture, with this solution we might get a delay in the send operation of event notification messages, say $message_i$, $message_{i+1}$ and so on, for new events scheduled by the current event execution. As pointed out in [3, 19], this might yield rollback thrashing due to delayed delivery of those messages at the recipient SOs (delay in the delivery is expected to increase the likelihood of timestamp order violations at the recipient SOs).

As another drawback, it is possible that non-minimal housekeeping time is required for tracking a message/antimessage, if any, revealing a causality inconsistency (the time interval length depends on the receipt order of messages/antimessages at the application level). This might obviously reduce the benefits from preemptive rollback due to reduced timeliness of rollback operations themselves.

Moreover, further strain on the execution, not reported in Figure 1.b, might be due to additional overhead required to support time inter-leaving itself.

In the next section we present a communication layer organization allowing efficient support of event preemptive rollback on a Myrinet cluster, with no need for inter-leaving between event execution and housekeeping. To achieve this, the layer design makes it aware of the message content, namely content-aware, which provides it with the ability to verify causality inconsistency of the running SO as soon as messages/antimessages arrive at the host, and prior those messages are delivered at the application level.

Actually, the work in [21] develops a content-aware layer for Myrinet clusters in support of parallel discrete event simulation. This layer offers functionalities for efficient dissemination of state information required by a particular type of optimistic synchronization to control the level of optimism in the execution, namely near-perfect state information synchronization [22]. That work differs from our solution since it does not tackle at all the problem of efficiently supporting event preemptive rollback operations.

To the best of our knowledge, there is a single simulation software supporting event preemptive rollback for optimistic parallel simulation, namely GTW [6]. This software has been developed for shared memory multiprocessor systems, and its ability to support event preemptive rollback resides in the fact that message passing between SOs does not rely on any standard message passing layer. Specifically, GTW implements message passing at the

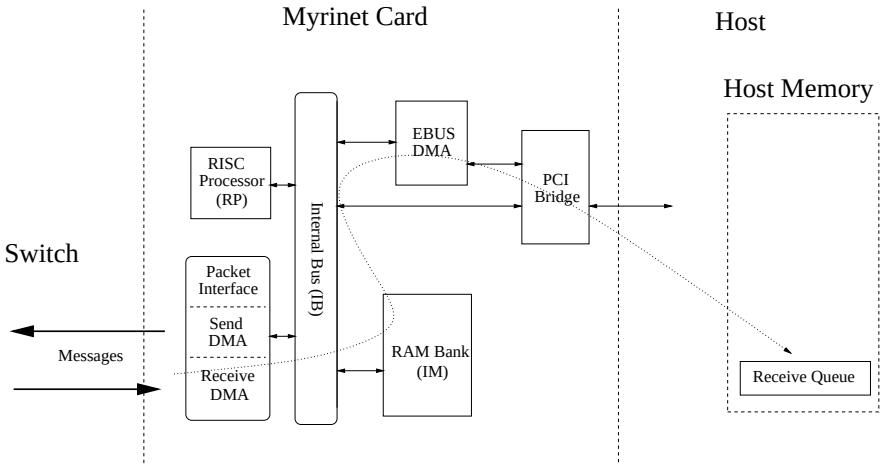

Figure 2: Myrinet Card Architecture.

simulation application level by directly manipulating the event list of the recipient SO. With this solution, the sender is able to determine the current simulation time for the recipient and whether any message/antimessage being sent to that SO violates causality. In the positive instance, the sender notifies the violation to the recipient which is allowed to timely interrupt any in-progress activity in order to execute rollback operations. Our solution is different in nature from the one implemented in GTW since we deal with parallel simulation on a distributed memory system, not a shared memory one.

3 Software Supports for Event Preemptive Rollback

The software supports we present are an extension of a classical message passing layer for Myrinet clusters. To make clear the extension design, we preliminary report a brief description of the Myrinet architecture and of a classical message passing layer organization. Then the extension is presented in details.

3.1 Myrinet Overview

Hardware Architecture. A Myrinet network consists of a switch based on wormhole technology and of interface cards in between the switch and the hosts, which are based on the LANai chip [13, 14, 15]. At high level, the interface card architecture can be schematized as shown in Figure 2. Specifically, it is a programmable communication device equipped with:

(a) An on-board programmable RISC Processor (RP).

(b) A RAM bank memory (Internal Memory, namely IM), accessible by RP, which can also be mapped into the host address space.

(c) A packet interface towards the switch, associated with Send/Receive DMA engines used for IM/packet-interface data transfer and vice versa.

(d) A PCI bridge towards the host.

(e) A DMA engine, called EBUS DMA (External Bus DMA) able to transfer data from the IM to the host main memory and vice versa.

All the components of the architecture are interconnected through an Internal Bus (IB) and the three DMA engines are programmable by RP. As a final observation, RP cannot access the host memory directly. Nonetheless, the program run by RP, typically known as *Control Program* (CP), can activate the EBUS DMA for performing data transfer operations to/from the host memory.

Classical Message Passing Layer Organization. In classical fast speed messaging layers for Myrinet, see for example [18], messages incoming from the switch are temporarily buffered into the IM, with data transfer between the packet interface and the IM taking place through the Receive DMA. The messages are then transferred into the receive queue, located onto host memory, through the EBUS DMA (see the directed dotted line in Figure 2). Once transferred into the receive queue, any message is received by the application program very efficiently by simply performing a `memcpy()` of the message content into a proper buffer in the application address space. In case multiple messages stored in contiguous slots of the IM must be transferred into the receive queue, a single EBUS DMA transfer is used for the whole set of messages. This technique is referred to as "block-DMA".

Another common choice consists of performing a send operation according to the "zero-copy" method, which means that the content of the message to be sent is directly copied into the IM. Then the message is transferred onto the network through the Send DMA. This optimization allows keeping the delivery latency at a minimum by avoiding additional buffering in host memory at the sender side.

The responsibility to program the three DMA engines anytime there is the need for supporting a given message transfer operation pertains to the CP run by RP. The main loop of the CP for our implementation is as follows:

```
1. While (1){
2.  if (message needs to be sent) activate_send_DMA();
3.  if (message needs to be received) activate_receive_DMA();
4.  if (Send DMA completed) complete_send();
5.  if (Receive DMA completed) complete_receive();
6.  if (block-DMA needed) block_DMA();
7. }
```

Reliability of message delivery is supported by an acknowledgment mechanism typically implemented at the level of the CP run by RP. In our implementation, acknowledgments are

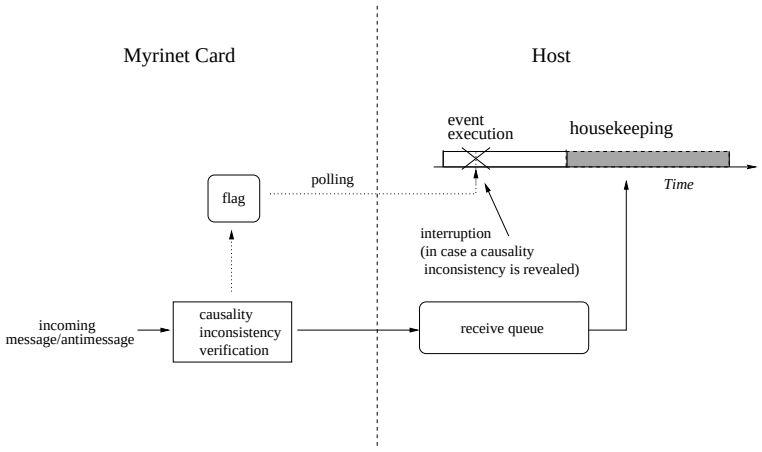

Figure 3: Causality Consistency Verification Scheme.

4 bytes packets managed through the functions `complete_send()` and `complete_receive()`. `complete_send()` takes also care of data re-transmission in case the acknowledgment does not arrive within a given timeout.

3.2 Layer Extension for Event Preemptive Rollback

We decided to extend the functionalities of a classical layer by actively employing the potential of RP. The aim of the extension is to introduce an on-the-fly causality inconsistency verification mechanism as schematized in Figure 3.

The idea is to check all the messages/antimessages incoming at the network level, in order to perform a causality inconsistency verification. This is done by comparing their timestamps with the current simulation clock of the recipient SO. In case the current simulation clock of the SO is greater than the timestamp of an incoming message/antimessage, a causality inconsistency is revealed. Notification of the inconsistency at the application level takes place by setting a flag whose value can be checked by the SO at its own pace during event execution.

Note that verification performed at the level of the CP run by RP can take place without altering the order in which messages/antimessages incoming from the switch are inserted into the receive queue located onto host memory. Like in our implementation, this order is typically FIFO, that together with the FIFO property guaranteed by transmission through the Myrinet switch, ensures point-to-point message delivery reflecting the message send order.

Actually, causality inconsistency verification can take place just after the incoming message/antimessage has been buffered into the IM on-board of the card (recall buffering is performed by the Receive DMA). At that point the message content is meaningful, and thus useful to construct a content-aware layer. Also, verification performed at that point does not require structural modifications to the CP, and does not require special handling, such as an additional intermediate buffering of the message/antimessage.

In our implementation, the function performing causality inconsistency verification has been called `peek()`, and the main loop of the CP has been slightly modified as shown below in order to embed a call to this function as soon as buffering of a message/antimessage into the IM is completed (see lines 5 to 7 of the loop):

```
1.  While (1){
2.   if (message needs to be sent) activate_send_DMA();
3.   if (message needs to be received) activate_receive_DMA();
4.   if (Send DMA completed) complete_send();
5.   if (Receive DMA completed){
6.       complete_receive();
7.       peek();
8.     }
9.   if (block-DMA needed) block_DMA();
10. }
```

To perform its task, the function `peek()` needs the current value of the simulation clock of the destination SO. This information must be made available into the IM since the CP cannot access host memory. To this purpose we have introduced, in the API associated with the layer, the function `start_checking(double simulation_clock)`, which writes the simulation clock of the SO hosted by the machine to a proper buffer located into the IM on-board of the card (recall IM is mapped in the host address space so that the host can perform read/write operations on it). Then the function `peek()` determines the occurrence of a causality inconsistency on that SO by simply comparing the current SO simulation clock and the timestamp of each incoming message/antimessage. If the timestamp value is lower than the simulation clock value, the current event execution is not causally consistent.

Beyond writing the simulation clock value to a proper buffer in the IM, `start_checking()` has also the responsibility to enable causality inconsistency verification performed at the level of the CP. This is done by setting a flag, namely `control_flag`, implemented as a word (4 bytes) located into the IM ([1]). Flag setting indicates to the function `peek()` that causality inconsistency verification is currently required by the application level. `peek()` checks the value of `control_flag` as its first action and avoids performing causality inconsistency verification in case the flag is not set. In other words, `start_checking()` acts as an initializer/activator for the causality inconsistency verification mechanism. This function should be called right after the simulation clock of the SO hosted by the machine is updated to the timestamp of the event that is going to be executed. In typical simulation code, this happens as soon as the SO enters the event routine. (A more detailed description of the usage, at the simulation application level, of the whole set of event preemptive rollback support functions belonging to the API will be presented in Section 3.4.)

[1] We have used a word instead of a single byte for `control_flag` since Myrinet cards internal circuitry is optimized for aligned words access.

Disabling causality inconsistency verification, i.e. resetting `control_flag`, can be performed through the function `stop_checking()`, which we have introduced in the API associated with the layer. This function should be called at the application level prior the start of any period in which causality inconsistency verification is not required (e.g. the execution period of housekeeping operations).

The reason why we have introduced `control_flag` is twofold. First, disabling causality inconsistency verification whenever not required by the application level increases the responsiveness of the CP in programming/controlling the DMA engines on-board of the card. Second, and more important, critical races between the software run at the host side and the CP run by RP might occur, causing errors in the outcome of the causality inconsistency verification mechanism. As we will discuss in Section 3.3, adequate management of `control_flag` as a three states flag prevents those races.

If a causality inconsistency is detected, the function `peek()` sets to TRUE a flag, namely `inconsistency_flag`, implemented as a word located into the IM. This flag is reset to FALSE by `start_checking()` upon its call. The simulation application software can check the value of `inconsistency_flag` through the function `check_causality_inconsistency()`, which we have introduced into the API. This function simply returns the flag value, namely TRUE in case a causality inconsistency has been revealed, FALSE otherwise. In other words, the function `check_causality_inconsistency()` can be used at the simulation application level for implementing an efficient polling-based approach to the detection of causality inconsistencies.

3.3 Tackling Critical Races

As already said, `control_flag` can be used as a mean to enable causality inconsistency verification performed at the level of the CP. We use it also as a mean to avoid critical races. To this purpose, `control_flag` is managed as a three states flag, where the values it can assume are OFF, ON and SHUT-DOWN.

Note that a critical race might occur in case access performed by the functions `start_checking()` and `peek()` to the content of the shared buffer located into the IM (which maintains the simulation clock of the SO hosted by the machine) is not guaranteed to be an atomic action. More precisely, the shared buffer is used for passing data between the host processor and the RP, therefore a critical race in the communication path between the functions `start_checking()` and `peek()` through the shared buffer may occur in case one or both the following conditions are verified:

(1) `start_checking()` does not write data into IM atomically.

(2) `peek()` does not read data into IM atomically.

Depending on host hardware features and on the specific Myrinet card, the conditions in points (1) and (2) might occur. Specifically, in case the PCI protocol works on a 32 bit bus,

writing the simulation clock of the SO into the IM cannot be guaranteed to be atomic since the simulation clock value is typically expressed (as in our implementation) with a double precision floating point. Also, in case the Myrinet device is equipped with a 32 bit Internal Bus - IB - (this is the case of the M2M-PCI32C Myrinet cards for which our implementation has been developed), then the function `peek()` run by RP needs two accesses on the IB to catch that simulation clock value.

To show an example of critical race occurrence, just due to non-atomicity of the write operation of simulation clock values into IM, let us first consider the sequence of statements associated with read/write operations performed by the functions `start_checking()`, `stop_checking()` and `peek()`, where `control_flag` is managed as a classical two states flag (i.e. an ON/OFF flag):

`start_checking(simulation_clock)`
A.1 write the four most significant bytes of `simulation_clock` into IM;
A.2 write the four least significant bytes of `simulation_clock` into IM;
A.3 set `inconsistency_flag` to FALSE;
A.4 set `control_flag` to ON;

`stop_checking()`
B.1 set `control_flag` to OFF;

`peek()`
C.1 if `control_flag` = ON
C.2 read the four most significant bytes of `simulation_clock` from IM;
C.3 read the four least significant bytes of `simulation_clock` from IM;
C.4 if message/antimessage timestamp < `simulation_clock`
C.5 set `inconsistency_flag` to TRUE;

Given that the software run at the host side and the function `peek()` execute asynchronously, the following sequence of read/write operations might occur:

1 A.1 `start_checking()` writes the four most significant bytes of `simulation_clock` into IM;
2 A.2 `start_checking()` writes the four least significant bytes of `simulation_clock` into IM;
3 A.3 `start_checking()` sets `inconsistency_flag` to FALSE;
4 A.4 `start_checking()` enables causality inconsistency checking by setting `control_flag` to ON;

5 C.1 `peek()` checks `control_flag`, with value ON;
6 C.2 `peek()` reads the four most significant bytes of `simulation_clock` from IM;

7 B.1 `stop_checking()` disables causality inconsistency tracking by setting `control_flag` to OFF;

8 A.1 `start_checking()` writes the four most significant bytes of `simulation_clock` into IM;
9 A.2 `start_checking()` writes the four least significant bytes of `simulation_clock` into IM;
10 A.3 `start_checking()` sets `inconsistency_flag` to FALSE;
11 A.4 `start_checking()` enables causality inconsistency checking by setting `control_flag` to ON;

12 C.3 `peek()` reads the four least significant bytes of `simulation_clock` from IM;
13 C.4 `peek()` compares the timestamp of the incoming message/antimessage with
the `simulation_clock` read from IM; *this simulation clock value is inconsistent*
14 C.5 `peek()` sets `inconsistency_flag` to TRUE in case of positive result of the comparison in line 13;

This sequence might produce an incorrect result since in line 13 the function `peek()` compares the timestamp of a message/antimessage with a simulation clock value whose 4 most/least significant bytes have been written by two different and subsequent activations of `start_checking()`. In other words, the value of `simulation_clock` read by `peek()` from IM results as the composition of two disjoint sets of bytes.

This problem derives from the fact that there exists the possibility of inter-leaving among read/write operations performed by the functions `start_checking()` and `peek()` on information in the shared buffer into the IM. In the example, the inter-leaving derives from multiple activations of the function `start_checking()` within the execution interval of the function `peek()`.

The following organization, with a three states management for `control_flag` allows execution of read/write operations on the shared buffer into the IM, performed by any single function, as an atomic action, with no possibility of inter-leaving:

```
start_checking(simulation_clock)
   A.0 wait until control_flag = OFF;
   A.1 write the four most significant bytes of simulation_clock into IM;
   A.2 write the four least significant bytes of simulation_clock into IM;
   A.3 set inconsistency_flag to FALSE;
   A.4 set control_flag to ON;

stop_checking()
   B.1 set control_flag to SHUT-DOWN;

peek()
   C.1 if control_flag = ON
   C.2    read the four most significant bytes of simulation_clock from IM;
   C.3    read the four least significant bytes of simulation_clock from IM;
   C.4       if message/antimessage timestamp < simulation_clock
   C.5          set inconsistency_flag to TRUE;
```

Figure 4 shows the state transition diagram for this solution. While `control_flag` is in the state OFF, `peek()` is not allowed to enter the block of statements from C.2 to C.4. Instead, `start_checking()` is allowed to enter the block of statements from A.1 to A.3. The converse occurs when the flag is in the state ON. On the other hand, when the flag is in the state SHUT-DOWN, both `peek()` and `start_checking()` are not allowed to enter their

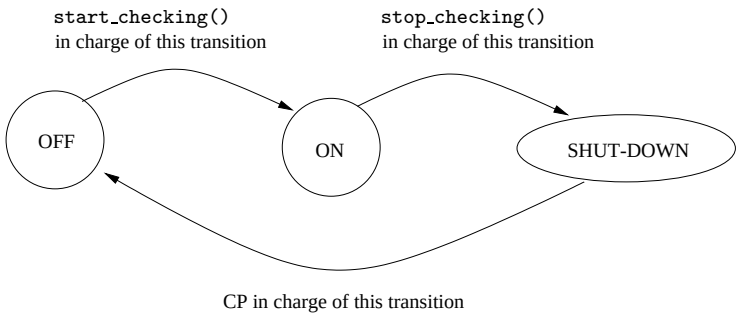

Figure 4: State Transition Diagram for `control_flag`.

statement sequences. To avoid live-lock of the application due to indefinite spin-locking of the function `start_checking()` around the SHUT-DOWN value of `control_flag` (see statement A.0), we have made an additional slight modification of the CP in order to let this program take care of transition of `control_flag` from SHUT-DOWN to OFF within its main loop. The final loop structure is therefore as follows (see line 9 for this last modification):

```
1. While (1){
2.   if (message needs to be sent) activate_send_DMA();
3.   if (message needs to be received) activate_receive_DMA();
4.   if (Send DMA completed) complete_send();
5.   if (Receive DMA completed){
6.      complete_receive();
7.      peek();
8.    }
9.   if (control_flag == SHUT_DOWN) control_flag = OFF;
10.  if (block-DMA needed) block_DMA();
11. }
```

3.4 API Usage

In this section we provide an overview on the usage of the `C` API that has been developed to efficiently support event preemptive rollback. Then we discuss the message format to be adopted at the application level.

3.4.1 Event Preemptive Rollback Support Functions

Figure 5 depicts how event preemptive rollback support functions are expected to be employed. As soon as the simulation clock of the SO is moved to the event to be executed (this typically happens at the event execution starting) the function `start_checking()` should be called. In case this function is called prior to updating the SO simulation clock, causality inconsistency verification might result less effective since it cannot reveal a timestamp order violation occurring in the simulation time interval between the not yet updated value of the simulation clock and the timestamp of the event currently executed by the SO.

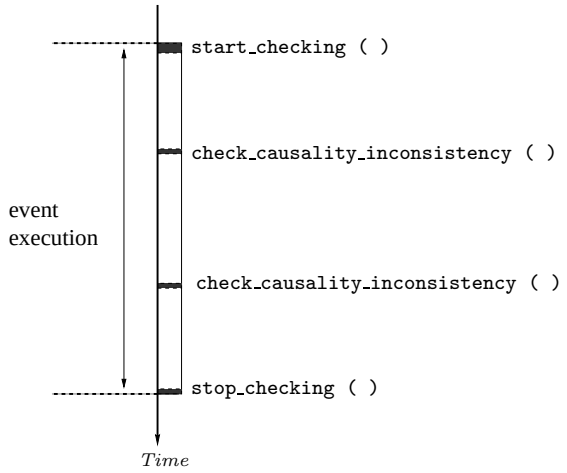

Figure 5: Pattern of Calls to Event Preemptive Rollback Support Functions.

Just before the event execution is completed, the function `stop_checking()` should be called to disable causality consistency verification. This also allows correct management of the state transition for `control_flag` since `control_flag` must necessarily pass through the SHUT-DOWN state before any further call to `start_checking()` is issued.

The two functions above initialize-enable and disable causality inconsistency verification. In order to read the verification outcome, periodic calls to `check_causality_inconsistency()` should be issued. In case one of these calls reveals a causality inconsistency, the event execution should be interrupted. As a last action before interruption, the function `stop_checking()` must be called to disable again the causality verification mechanism until a new call to `start_checking()` is issued. We also note that the pathological scenario depicted in the Introduction, where event processing routines go into an infinite loop in case of causality violations [17], can be effectively prevented by combining our software facilities with Operating System timing events, such as `SIGALRM` on UNIX systems. This kind of events (periodically) return the control to a signal handler, thus providing the opportunity to eventually invoke `check_causality_inconsistency()`.

We have measured the time required to execute `check_causality_inconsistency()` in case of a Pentium II 300 MHz with LINUX (kernel version 2.0.32) equipped with an M2M-PCI32C Myrinet card ([2]), and the measurement outcome is 0.5 μsecs. Considering that parallel discrete event technology is typically adopted for simulation models with event granularity (i.e. event execution time) ranging from several tens or hundreds of microseconds up to some milliseconds or more on current conventional architectures ([3]), the overhead due to polling performed by `check_causality_inconsistency()` is negligible in practice even if

[2]The data we report in Section 4 refer to experiments executed on a Myrinet cluster of this type of PCs.

[3]For models with lower event granularity that is, few tens of microseconds or less, it would be preferable to run multiple copies of the simulation sequentially on the available machines as a different, more convenient, form of parallelism.

several calls to this function are performed within a single event execution. With respect to the strategy for placing these calls within the event code, we note that revealing a causality inconsistency right before scheduling any new event (either locally or destined to a remote SO) is likely to amplify the benefits from preemptive rollback.

3.4.2 Message Format

From the point of view of pure communication, the Myrinet layer we have extended with previously described functionalities is characterized by the following set of `C` API functions. (i) `send(int msg_type, int machine_id, void *message)`, where `msg_type` defines the type of the message, `machine_id` defines the destination host for the message, and `message` is a pointer to the memory area containing the data to be sent (in the current implementation the number of bytes that can be sent via a single message is determined at compile time of the layer so that it can be tailored to the requirements of the overlaying application); this function passes the data to be transmitted to the network card employing the "zero-copy" optimization described in Section 3.1. (ii) `receive(int msg_type, void *message, int *machine_id)`, which is a receive function that returns a message of a given type, if any, in the memory area pointed by `message`, and also the identifier of the sender host in the memory area pointed by `machine_id`; this function copies the message content from the receive queue, to the application address space through a simple `memcpy()` call.

Send and receive functions show that the message content is managed as a simple sequence of bytes, therefore the overlaying application is in charge of defining an adequate structure for the content itself. On the other hand, the layer extension we have presented accesses the content, i.e. the timestamp, at the level of the CP through the function `peek()`. To make this access working properly, any message/antimessage must comply with the following format: the first eight bytes must contain the timestamp, expressed as a double precision floating point represented as a little endian value (this corresponds to the INTEL CPU representation). The rest of the message is of no concern to the CP.

Note that the causality inconsistency verification mechanism we have implemented does not need to discriminate between messages and antimessages, therefore the flag indicating whether the incoming data refer to a simulation message or a simulation antimessage either can be specified by `msg_type` or can be placed wherever in the data content, except the first 8 bytes that, as said before, are reserved for the timestamp value.

4 Experimental Results

In this section we report the results of an empirical study on the effectiveness of the software supports for event preemptive rollback we have implemented. As a testbed we have used a Personal Communication System (PCS) simulation application. We first present the testing

environment and the specific PCS simulation model we have used. Then we introduce the employed evaluation metrics and report the obtained results.

4.1 Testing Environment

The experiments were all performed on a cluster of 8 Pentium II 300 MHz (128 Mbytes RAM - 512 Kbytes second level cache). PCI on these machines works at 32 bits, with frequency 33 MHz. All the PCs of the cluster run LINUX (kernel version 2.0.32) and are equipped with M2M-PCI32C Myrinet cards. Also, the machines are interconnected through a 1.2 GHz Myrinet switch.

The message size we have used at the communication layer level (i.e. the layer packet size) is 40 bytes, which fits the requirements of the selected test case. For message size 40 bytes, the delivery delay, intended as the time to transfer a message of 40 bytes between two application address spaces on distinct machines in the cluster, is on the order of 20 microseconds when no congestion occurs on the switch. Such a latency value is aligned with the latency achievable by other Myrinet tailored message passing layers for the same message size, when considering an architecture with features similar to the one we have employed [18]. Actually, results in [16, 20, 24] report latencies lower than the one we have observed (i.e. on the order of 10 microseconds), however these measurements have been taken on architectures more advanced than the architecture we have employed. Specifically, all those performance studies are related to architectures with 64 bits, 66 MHz PCI bus. In addition, they employ a 2 GHz Myrinet switch.

With respect to other main features of the software being used, we remark that memory space for new entries into the event lists of the SOs is dynamically allocated by using classical `malloc()` calls, and the event lists are implemented as simple linked lists. Finally, the cancellation phase is implemented following the aggressive strategy, that is antimessages are sent as soon as the SO enters rollback operations [8].

4.2 The PCS Simulation Model

In a PCS system, base stations provide communication services to mobile units. In our simulation model the service area (coverage area) is partitioned into cells that are represented as hexagons, therefore all the cells, except bordering cells of the coverage area, have six neighbors. Each SO models a cluster of cells of a given portion of the coverage area.

A cell represents a receiver/transmitter having a fixed number of 100 channels allocated to it. The model is call-initiated [5] since it only simulates the behavior of a mobile unit during conversation, i.e. the movement of a mobile unit is not tracked unless the unit itself is in conversation. Therefore, the model is organized around two entities, namely cells and calls. Call requests arrive to each cell according to an exponential distribution [2, 4, 5] with inter-arrival time t_{int} seconds.

For each cell, the corresponding SO maintains statistics, information about busy channels and, for each channel, information about features of the mobile unit involved in the ongoing call (e.g. scheduled call termination time, call initiation time, class of the mobile unit, etc.), if any. As a result, state information maintained for each cell has size of about 4 Kbytes ([4]). Checkpointing to support state recovery in case of rollback is performed incrementally [1, 23, 25], with the meaning that upon the occurrence of a simulation event at a given SO, the state information saved is only the one related to the cell whose state is going to be updated by the execution of the event.

There are three types of events, namely hand-off, due to mobile unit cell switch, call termination and call arrival. A call termination simply involves the release of the associated channel, whose identifier is maintained into the event compound structure, and statistics update. A call arrival checks if there is at least an available channel. In the negative instance, the incoming call is simply counted as a block, otherwise an available channel is allocated for the call and signal-to-interference-plus-noise ratio is computed in order to determine the power for the transmission on that channel [11]. The cost of this operation is dependent on the amount of busy channels at the time of the call arrival. When a hand-off occurs between adjacent cells, the hand-off event at the cell left by the mobile simply involves the release of the channel and statistics update. Instead, the hand-off event at the destination cell checks for channel availability, and allocates an available channel, if any, for the call. If there is no available channel, then the call is simply cut off (dropped). Otherwise, signal-to-interference-plus-noise ratio calculation is required.

In our model there are two distinct classes of mobile units. Both of them are characterized by a residence time within a cell which follows an exponential distribution [2], with mean 3 minutes (fast movement units) and 30 minutes (slow movement units), respectively. The average holding time for each call associated with both fast and slow movement units is 2 minutes. When a call arrives at a cell, the type (slow or fast) of the mobile unit associated with the incoming call is selected from a uniform distribution, therefore any call is equally likely to be destined to a fast or a slow movement mobile unit.

In the experimental study we have used eight SOs, each one hosted by a distinct machine in the cluster. Also, each SO simulates a coverage area consisting of 25 cells, therefore the model size, in terms of total amount of simulated cells for the coverage area, has been set at 200.

We have studied the effects of event preemptive rollback while varying the channel utilization factor. Specifically, we have varied the call interarrival time per cell t_{int} between 12 and 1.6 seconds. Given that the average call holding time is 2 minutes, and the number of channels per cell is fixed at 100, variation of t_{int} in that range gives rise to channel utilization

[4]PCS models might exhibit smaller granularity for state information associated with a cell [4, 5]. This might happen, for example, when state information maintained for each cell is almost exclusively related to the amount of busy channels.

factors ranging in the interval between 10% and 75%.

Actually, varying the channel utilization factor produces a variation in the granularity of simulation events since, as already pointed out, the cost of signal-to-interference-plus-noise ratio calculation depends on the amount of busy channels at the time of the call arrival. Specifically with a lower value for the utilization factor, the expected number of busy channels upon call arrival is kept low, therefore we get relatively low granularity for events that require channel allocation and signal-to-interference-plus-noise ratio calculation. On the other hand, we have longer expected execution time for these events in case of higher values of the utilization factor since we expect a non-minimal amount of busy channels upon call arrival.

Note that variation of the event granularity allows us to observe the effects of event preemptive rollback for a wide spectrum of communication frequencies between SOs (the larger the event granularity, the lower the communication frequency). Additionally, variation in the event granularity often produces a variation in the rollback pattern. Therefore we can also observe the effects of preemptive rollback with different patterns of rollback.

All the events in the simulation invoke the `check_causality_inconsistency()` primitive before scheduling any new event. In addition, call arrival events and hand-off events at a destination cell (i.e. the coarser grain events in the simulation) invoke `check_causality_inconsistency()` also on a periodic basis during the event computation, namely each 200 μsec.

4.3 Evaluation Metrics

We report measures related to the following parameters:

- The *event rate* (ER), that is, the number of committed simulation events per second. This parameter indicates how fast is the simulation execution, it is therefore representative of the achieved performance.

- The *rollback frequency* (RF), that is, the ratio between the number of rollbacks and the total number of executed simulation events. This parameter is an indicator of the potential for the timeliness in the execution of rollback procedures to positively impact performance by reducing the amount of rollback occurrences per simulation event.

- The *revoked events per committed event* (RECE), that is the ratio between total number of events that have been revoked and the total number of committed events. Actually, RECE keeps track both of events scheduled for other SOs and revoked through the corresponding antimessages, and of the events scheduled by a SO for itself, which also must be revoked but with no need for sending antimessages over the network. Therefore this parameter is an indicator of both (i) the additional event list management cost and (ii) the communication cost to be paid due to the rollback mechanism, evaluated for each productive simulation event execution (i.e. for each committed event).

We have measured the previous parameters both for the case of a typical execution with no preemptive rollback and for the case of event preemptive rollback. For the case of event preemptive rollback we have also measured the fraction F of rollback occurrences that have been started through event preemption. This parameter indicates the effectiveness of preemptive rollback in revealing causal inconsistencies during event execution. Additionally, we report the speedup achieved with event preemptive rollback over a sequential execution of the same simulation model on one of the machines of the cluster.

Each reported value results as the average over 10 runs, all done with different random seeds for the stochastic timestamp increment. At least 2×10^6 committed events were simulated in each run. (Although not reported we have obtained confidence intervals less than 10% at the 90% confidence level on all the average values.)

4.4 Results

In Figure 6, the ER values are plotted. The data show that event preemptive rollback allows an increase in the execution speed on the order of 11-13%, independently of the channel utilization factor, i.e. independently of the event granularity. By the plots for the other observed parameters, we have several sources for the increase in the execution speed. By the results in Figure 7, event preemptive rollback slightly reduces RF values, which shows the potential of event preemptive rollback itself in reducing the cascading rollback phenomenon. As a consequence, we get a reduction of the cost due to rollback procedures, since they are activated less frequently. This happens especially when the rate of rollbacks in the parallel execution is higher, i.e. when the channel utilization factor is lower. Additionally, by the results in Figure 8, event preemptive rollback reduces the amount of revoked events per committed event, namely RECE, which means (i) a reduction in the communication cost (due to a reduction of the amount of messages to be revoked, and a reduction of the corresponding antimessages) and also (ii) a reduction in the event list management cost since a lower amount of events are inserted in, and then extracted by, the event list due to the rollback mechanism.

We note that the outcoming reduction in the communication cost and in the event list management cost is expected to have more relevant impact on fine grain computations (i.e. when the channel utilization factor is low) as compared to coarse grain ones. On the other hand, for coarse grain computations, namely high channel utilization factor, the interruption in the execution of an event due to event preemptive rollback has the potential to produce relevant reduction of the waste of CPU time due to processing of causally inconsistent events. This is the reason why when the channel utilization factor is increased, the performance gain due to event preemptive rollback does not decrease even if the gains in the parameters RF and RECE get reduced.

The plot for F in Figure 9 is an additional empirical support to the fact that, for high channel utilization factor (i.e. large event granularity), a large source of gain of event preemp-

tive rollback actually derives from early interruption of causally inconsistent events that are costly to simulate. Specifically, although F assumes larger values for lower channel utilization factor, it remains on the order of 0.27 even when the channel utilization factor reaches its maximum value of 75%. This means that at least 27% of the rollback occurrences produce event preemption for timely activation of rollback procedures. This phenomenon, combined with the larger granularity of events that are preempted, actually yields strong reduction of the cost due to the rolled back computation.

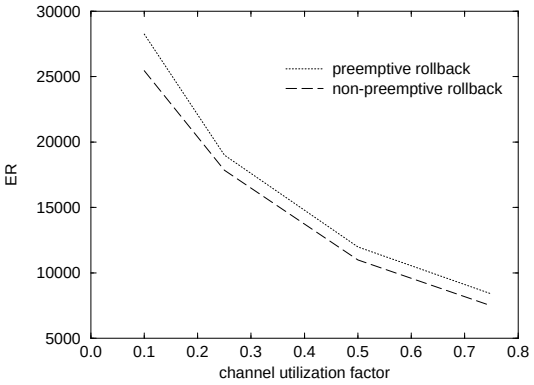

Figure 6: ER vs the Channel Utilization Factor.

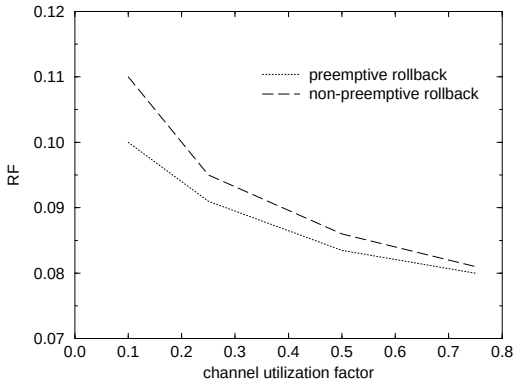

Figure 7: RF vs the Channel Utilization Factor.

Finally, we note that the speedup achieved through event preemptive rollback on 8 machines ranges between 4.7 and 6.4 (see Figure 10), which is therefore in between 60% and 80% of the ideal one. Recalling that speedup on the order of 50% of the ideal one is considered as a classical threshold determining acceptable performance in the parallel execution of a discrete event simulation model on distributed memory systems, we can conclude that the results we have reported have been obtained for a scenario in which parallel execution is supported

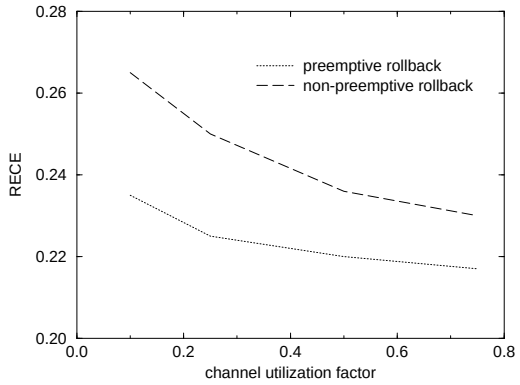

Figure 8: RECE vs the Channel Utilization Factor.

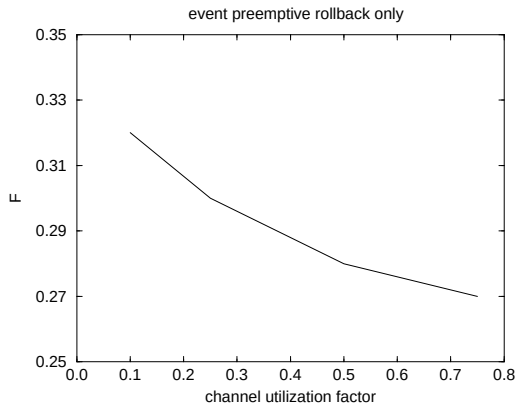

Figure 9: F vs the Channel Utilization Factor.

effectively.

5 Summary

In this paper we have presented software supports for event preemptive rollback operations relying on timely interruption of the execution of a causally inconsistent simulation event. These supports have been developed for optimistic parallel discrete event simulators running on top of Myrinet based architectures, and have been derived from an extension of a classical Myrinet message passing layer. We have discussed how to use the event preemptive rollback functions at the simulation application level and we have also reported the results of an empirical study on a personal communication system simulation application, which demonstrate the effectiveness of the presented extension. Specifically, these results show that the performance can be increased up to 11-13% for both fine and coarse event configurations.

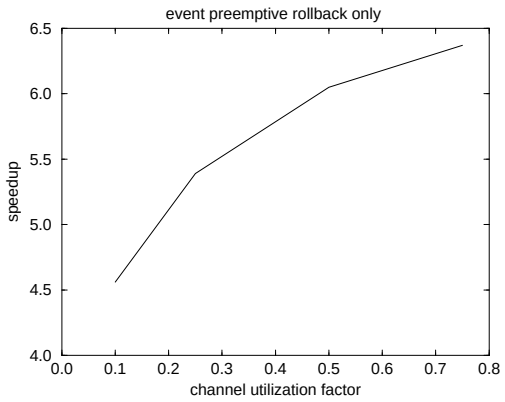

Figure 10: Speedup vs the Channel Utilization Factor.

References

[1] H. Bauer and C. Sporrer, "Reducing Rollback Overhead in Time Warp Based Distributed Simulation with Optimized Incremental State Saving", *Proc. 26th Annual Simulation Symposium*, pp.12-20, April 1993.

[2] A. Boukerche, S.K. Das, A. Fabbri and O. Yildz, "Exploiting Model Independence for Parallel PCS Network Simulation", *Proc. 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pp.166-173, May 1999.

[3] C.D. Carothers, R. Fujimoto and P. England, "Effect of Communication Overheads on Time Warp Performance: An Experimental Study", *Proc. 8th Workshop on Parallel and Distributed Simulation (PADS'94)*, pp. 118-125, July 1994.

[4] C.D. Carothers, D. Bauer and S. Pearce, "ROSS: a High Performance, Low Memory, Modular Time Warp System", *Proc. 14th Workshop on Parallel and Distributed Simulation (PADS'00)*, pp.53-60, May 2000.

[5] C.D. Carothers, R. Fujimoto and Y.B. Lin, "A Case Study in Simulating PCS Networks Using Time Warp", *Proc. 9th Workshop on Parallel and Distributed Simulation (PADS'95)*, pp.87-94, June 1995.

[6] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: A Time Warp System for Shared Memory Multiprocessors", *Proc. 1994 Winter Simulation Conference*, pp. 1332-1339, December 1994.

[7] R.M. Fujimoto, "Parallel Discrete Event Simulation", *Communications of ACM*, Vol.33, No.10, pp.30-53, 1990.

[8] A. Gafni, "Space Management and Cancellation Mechanisms for Time Warp", *Tech. Rep. TR-85-341*, University of Southern California, Los Angeles (Ca,USA).

[9] RAMIX, "Intelligent Ethernet Interface Solutions". Available at `http://www.ramix.com`

[10] D.R. Jefferson, "Virtual Time", *ACM Transactions on Programming Languages and Systems*, Vol.7, No.3, pp.404-425, 1985.

[11] S. Kandukuri and S. Boyd, "Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications", *IEEE Transactions on Wireless Communications*, Vol.1, No.1, pp.46-55, 2002.

[12] Myricom, "Myrinet Overview". Available at `http://www.myri.com/myrinet/overview/index.html`

[13] MYRICOM, "LANai 4", *Draft*, February 1999.

[14] MYRICOM, "LANai 7", *Draft*, June 1999. Available at `http://www.myri.com/vlsi/LANai7.pdf`

[15] MYRICOM, "LANai 9", May 2001. Available at `http://www.myri.com/vlsi/LANai9.pdf`

[16] MYRICOM, "GM 1.6.4 API Performance with PCI64B and PCI64C Myrinet/PCI Interfaces", April 2003. Available at `http://www.myri.com/myrinet/performance/index.html`.

[17] D.M. Nicol and X. Liu, "The Dark Side of Risk (what your mother never told you about Time Warp)", *Proc. 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pp.188-195, June 1997.

[18] S. Pakin, M. Lauria and A. Chen, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet", *Proc. 9th ACM Int. Conference on Supercomputing*, 1995.

[19] B.R. Preiss, W.M. Loucks and D. MacIntyre, "Effects of the Checkpoint Interval on Time and Space in Time Warp", *ACM Transactions on Modeling and Computer Simulation*, Vol. 4, No. 3, pp.223-253, 1994.

[20] L. Prylli, C. Pham and B. Tourancheau, "A Software Suite for High-Performance Communications on Clusters of SMPs", *Cluster Computing*, Vol.5, No.4, pp.353-363, 2002.

[21] S. Srinivasan, M.J. Lyell, P.F. Reynolds, Jr. and J.W. Wehrwein, "Implementation of Reductions in Support of PDES", *Proc. 12th Workshop on Parallel and Distributed Simulation (PADS'98)*, pp.116-123, May 1998.

[22] S. Srinivasan and P.F. Reynolds Jr., "Elastic Time", *ACM Transactions on Modeling and Computer Simulation*, Vol.8, No.2, pp.103-139, 1998.

[23] J. Steinman, "Incremental State Saving in SPEEDES Using C Plus Plus", *Proc. 1993 Winter Simulation Conference*, pp.687-696, December 1993.

[24] H. Tezuka, A. Hori, Y. Ishikawa and M. Sato, "PM: An Operating System Coordinated High Performance Communication Library", *Proc. International Conference and Exhibition on High-Performance Computing and Networking*, pp.708-717, April 1997.

[25] B.W. Unger, J. Cleary, A. Covington and D. West, "External State Management System for Optimistic Parallel Simulation", *Proc. 1993 Winter Simulation Conference*, pp.750-755, Decempber 1993.