

# Modeling and Optimization of Non-Blocking Checkpointing for Optimistic Simulation on Myrinet Clusters

Francesco Quaglia and Andrea Santoro  
Dipartimento di Informatica e Sistemistica  
Università di Roma "La Sapienza"  
Via Salaria 113, 00198 Roma, Italy  
{quaglia,santoro}@dis.uniroma1.it

## ABSTRACT

Checkpointing and Communication Library (CCL) is a recently developed software implementing CPU offloaded checkpointing functionalities in support of optimistic parallel simulation on myrinet clusters. Specifically, CCL implements a *non-blocking* execution mode of memory-to-memory data copy associated with checkpoint operations, based on data transfer capabilities provided by a programmable DMA engine on board of myrinet network cards. Re-synchronization between CPU and DMA activities must sometimes be employed for several reasons, such as maintenance of data consistency, thus adding some overhead to (otherwise CPU cost-free) non-blocking checkpoint operations. In this paper we present a cost model for non-blocking checkpointing and derive a performance effective re-synchronization semantic which we call *minimum cost re-synchronization (MC)*. With this semantic, an occurrence of re-synchronization either commits an on-going DMA based checkpoint operation (causing suspension of CPU activities) or aborts the operation (with possible increase in the expected rollback cost due to a reduced amount of committed checkpoints) on the basis of a minimum overhead expectation evaluated through the cost model. We have implemented *MC* within CCL, and we also report experimental results demonstrating the performance benefits from this optimized re-synchronization semantic, in terms of increase in the execution speed, for a Personal Communication System (PCS) simulation application.

## Categories and Subject Descriptors

C.4 [Performance of Systems]: Design Studies; C.2.4 [Computer-Communication Networks]: Distributed Systems—*distributed applications*; D.4.2 [Operating Systems]: Storage Management—*main memory*; I.6.7 [Simulation and Modeling]: Simulation Support Systems—*environments*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'03, June 23–26, 2003, San Francisco, California, USA.  
Copyright 2003 ACM 1-58113-733-8/03/0006 ...\$5.00.

## General Terms

Design, experimentation, performance

## Keywords

Optimistic simulation, checkpointing, DMA, performance optimization

## 1. INTRODUCTION

Parallel discrete event simulation [11] is based on partitioning the simulation model into a set of Logical Processes (LPs) which simulate distinct parts of the system under investigation. The scheduling of simulation events among the LPs takes place through the exchange of messages carrying the content and the occurrence time (timestamp) of the events. To ensure correct simulation results, synchronization mechanisms must be used to maintain a non-decreasing timestamp order for the execution of simulation events at each LP.

Optimistic mechanisms [13] allow each LP to execute events whenever they are available, thus performing no preventive verification on whether the execution itself meets the correctness criterion. On the other hand, if a timestamp order violation is detected, a rollback procedure recovers the LP state vector to a previous correct value. This is done by exploiting records of state information, namely *checkpoints*, taken during the parallel execution. Specifically, state recovery is accomplished by reloading the latest checkpoint preceding, or coinciding with, the state to be recovered, and re-updating state variables within the state vector through the replay of intermediate events, if any. The re-update phase is commonly termed *coasting-forward* (<sup>1</sup>). By their nature, optimistic synchronization mechanisms allow strong exploitation of parallelism as demonstrated by large speedups, achieved against classical sequential execution, for simulations of personal communication systems [5, 6], queuing networks [10], logic circuits [2], and aviation control systems [36], among others.

Since rollback is an endemic phenomenon in optimistic simulation, fast access to checkpointed state information is

<sup>1</sup>Another way to support state recovery is to take checkpoints of portions of the LP state vector incrementally, and to backward apply the incremental logs upon rolling back [3, 26, 34, 35]. However, this approach is typically less common primarily due to reduced transparency at the application programmer level.

mandatory in order to keep state recovery costs at acceptable levels. As a consequence checkpoints of the LP state vector are maintained into volatile memory buffers in the address space of the simulation application program. Actually, the size of LP state vectors can reach up to several Kbytes, e.g. simulations of mobile systems [26]. Also, LP state vectors might need to be saved relatively frequently for reducing the overhead of coasting forward, especially in case rollbacks occur frequently in the parallel execution and the granularity of simulation events is non-minimal [9, 15, 21, 25]. Therefore, charging on the CPU memory-to-memory data copy associated with checkpointing might result in a non-negligible cost.

In an attempt to avoid the CPU overhead of any single checkpoint operation, a Checkpointing and Communication Library (CCL) providing CPU offloaded checkpointing functionalities has been recently implemented in support of optimistic simulation on myrinet clusters [23], i.e. a Commercial Off-The-Shelf (COTS) architecture recognized as a standard for parallel computing applications. In CCL a checkpoint operation (i.e. memory-to-memory transfer of the LP state vector into the checkpoint buffer) is charged on a programmable DMA engine on board of myrinet network cards, thus allowing the CPU to carry out other simulation specific operations while checkpointing is in progress. In other words, CCL implements an innovative, *non-blocking* execution mode of checkpointing <sup>(2)</sup>.

Although this solution has shown the potential for significantly improving the execution speed of the parallel simulation application [23, 28], an important performance issue to address is related to the fact that the CPU and the DMA engine must sometimes “re-synchronize” due to a set of reasons. Just to name one, any LP scheduled for event execution cannot proceed while a DMA based checkpoint operation involving its state vector is in progress, otherwise that checkpoint might result in an incorrect snapshot of the LP state due to updates issued by the LP on the state vector while executing the event.

Re-synchronization has been implemented within CCL according to a *Conditional Checkpoint Abort (CCA)* semantic [23], which causes freezing of simulation activities carried out by the CPU only in case at least a threshold fraction of the state vector currently being checkpointed through DMA has been already transferred into the checkpoint buffer (simulation activities are resumed as soon as the checkpoint operation gets completed). In the opposite case, the checkpoint operation is aborted, and simulation activities on the CPU are allowed to immediately proceed. By carefully tuning the threshold fraction [23], relatively adequate trade-off between checkpointing overhead (due to freezing associated with checkpoint commitment) and coasting-forward overhead (due to uncommitment of scheduled checkpoints) can be achieved by *CCA*.

In this paper, we present a detailed cost model for non-blocking checkpointing, and derive a *Minimum Cost (MC)*

<sup>2</sup>Non-blocking checkpointing has been widely explored in the context of fault tolerance [7, 14, 20], with the meaning that the application is allowed to proceed while process state information is being transferred onto stable storage, e.g. a network file server. It was never explored for the case of log of state information maintained into volatile memory buffers of a given application program. This is where the innovation of non-blocking checkpointing supported by CCL resides.

re-synchronization semantic based on the model, which takes the checkpoint commit/abort decision on the basis of a minimum overhead expectation. Specifically, the cost model associates with any re-synchronization point a checkpointing-recovery overhead, expressed for both the cases of commit and abort of the on-going checkpoint operation. The convenience of checkpoint commit/abort is then established by *MC* solving the cost model. In order to solve the model, we need an estimate of the residual completion latency for an on-going non-blocking checkpoint operation upon the occurrence of re-synchronization. While presenting software extensions to support *MC*, we show how this parameter can be estimated at low cost in the specific implementation of non-blocking checkpointing within CCL.

Note that, although both *CCA* and *MC* are aimed at improving performance of non-blocking checkpointing by committing/aborting an on-going checkpoint operation at the re-synchronization point, they are totally different in nature. Specifically, in *CCA* the criterion for taking the commit/abort decision, namely the threshold percentage for the on-going checkpoint operation to be passed to the function implementing re-synchronization, is predetermined upon re-synchronization occurrence. Instead, *MC* is based on a criterion which is not predetermined, namely the result of the evaluation of the checkpointing-recovery cost model upon re-synchronization occurrence. In other words, the approach we take in this paper addresses the performance of non-blocking checkpointing from an innovative perspective.

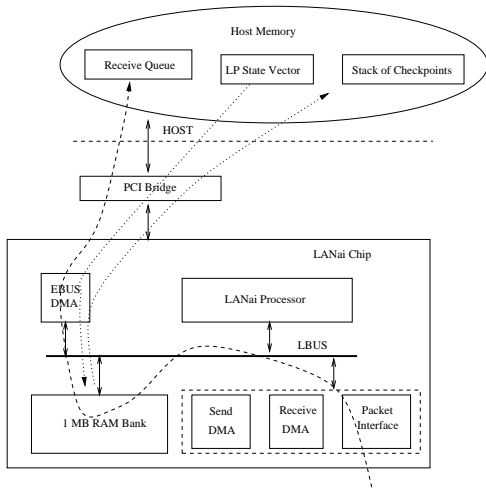
We also report the results of an experimental study, carried out on a Personal Communication System (PCS) simulation application, showing that *MC* improves the execution speed up to 7% as compared to the previous re-synchronization semantic, and, more important, up to 13% as compared to classical optimized checkpointing techniques based on CPU charged (blocking) checkpoints. Actually, the gain obtained through *MC* allows the achievement of speedup (over sequential execution of the same simulation model) in the order of up to 78% of the ideal one on a myrinet cluster of eight LINUX machines.

The remainder of this paper is organized as follows. Section 2 provides an overview of CCL. The cost model and the *MC* re-synchronization semantic, including implementation details, are presented in Section 3. The results of the experimental study are reported in Section 4.

## 2. OVERVIEW OF CCL

CCL has been designed for the M2M-PCI32C myrinet card, based on the LANai 4 chip [17]. This chip, whose high level structure is schematized in Figure 1, is a programmable communication device consisting of: (A) An internal bus, namely LBUS (Local BUS). (B) A programmable processor connected to the LBUS, which we will refer to as LANai processor. (C) A RAM bank of 1 Mbyte (LANai internal memory), connected to the LBUS, which can be mapped into the memory address space of the host. (D) A packet interface between the myrinet switch and the LANai chip, accessible by the LANai processor. (E) Three DMA engines used respectively for: (i) packet-interface/internal-memory transfer (Receive DMA), (ii) internal-memory/packet-interface transfer (Send DMA), (iii) internal-memory/host-memory transfer or vice-versa (EBUS DMA, namely External Bus DMA).

Host access to the LANai internal memory takes place



**Figure 1: High level structure of the M2M-PCI32C card and EBUS DMA data transfers.**

through a PCI bridge, which is also used for EBUS DMA data transfer from the host memory to the LANai internal memory and vice versa.

Communication functionalities provided by CCL fully exploit the potential offered by the hardware components on the network card. As in the common choice to fast speed messaging layers for myrinet (see for example [18]), messages incoming from the network are temporarily buffered into the LANai internal memory (data transfer between the packet interface and the internal memory takes place through the Receive DMA) and then transferred into the receive queue, located onto host memory, through the EBUS DMA (see the directed dashed line in Figure 1). Given that the message is already in the host memory when performing a receive operation, this operation does not involve access to the PCI bridge, thus keeping at a minimum the overhead of any message receipt.

A classical optimization called “block-DMA” is used to transfer incoming messages from the LANai internal memory to the host memory. This optimization allows incoming messages stored in contiguous message slots of the LANai internal memory to be transferred using a single EBUS DMA operation. Following another common design choice, any send operation issued by the application involves copying the message content directly into the LANai internal memory. This is also referred to as “zero-copy” send. Then the message is transferred onto the network through the Send DMA. This optimization allows keeping the delivery latency at a minimum by avoiding intermediate buffering at the sender side.

The EBUS DMA is used not only to transfer messages from the LANai internal memory to the receive queue, but also for data transfer associated with checkpointing. Specifically, a checkpoint operation involves data transfer from the LP state buffer (located onto host memory) to the stack of the checkpointed state vectors of the LP (also located onto host memory). As shown by the directed dotted lines in Figure 1, the transfer operation is charged to the EBUS DMA that uses the LANai internal memory as a temporary buffer<sup>(3)</sup>.

<sup>3</sup>Temporary buffering is needed since the EBUS DMA does

The responsibility to program the three DMA engines anytime there is the need for supporting a given data transfer operation pertains to a *control program* run by the LANai processor. Therefore, issuing a checkpoint request at the application level actually means requesting the LANai processor to program the EBUS DMA for the data transfer operation. This is done by the function `non_block_ckpt(int LP_id, double simulation_clock)`, included in the API provided by CCL, where `LP_id` is the identifier of the LP whose state vector needs to be checkpointed, and `simulation_clock` is the value of the current simulation time of the LP<sup>(4)</sup>.

Any checkpoint operation is split by the control program into a sequence of data transfer operations to be performed by the EBUS DMA. Each operation transfers up to a maximum amount of bytes, called *burst*, from the LP state vector to the LANai internal memory (intermediate buffering) or from the LANai internal memory to the stack of checkpoints of the LP. Also, lower priority is assigned to data transfer associated with checkpointing as compared to block-DMA. These two engineering choices allow checkpointing functionalities offered by CCL to produce negligible interference with communication functionalities. Specifically, splitting any checkpoint operation into a sequence of bursts allows prompt re-assignment of the hardware resources on board of the myrinet card (i.e. the EBUS DMA, the PCI bridge and the LBUS) to communication operations due to their higher priority<sup>(5)</sup>.

Re-synchronization between CPU activities and checkpointing activities carried out by the EBUS DMA is required to avoid data inconsistency (i.e. updating an LP state vector that is still being transferred in the checkpoint buffer through the EBUS DMA) and to prevent contention on the hardware due to activation of multiple checkpoint requests. As discussed in [23, 24] handling contention through time-sharing or queuing might reduce responsiveness of the control program run by the LANai processor as respect to the activation of DMA functionalities for supporting message transfer, with possible performance degradation of communication functionalities. The re-synchronization functionality should be activated whenever the LP that lastly issued a checkpoint request is re-scheduled for event execution (this prevents data inconsistency) and also before issuing any new checkpoint request (this prevents hardware contention).

As pointed out in the Introduction, the re-synchronization functionality offered by CCL is based on the *CCA* semantic [23]. To implement this semantic, CCL maintains a counter, namely `completed_transfers`, in the LANai internal memory, which is managed as follows. The counter is reset by the `non_block_ckpt()` function upon issuing a checkpoint

not support host-memory to host-memory data transfer directly. It only supports host-memory to internal-memory transfer or vice versa.

<sup>4</sup>CCL manages the checkpoint stacks of the LPs in a totally transparent way to the application programmer [23], which is the reason why `LP_id` is a sufficient parameter to identify both the state buffer and the entry into the stack of checkpoints that must be involved in the data transfer.

<sup>5</sup>[24] shows how to identify the maximum value for the burst length, which allows the performance of communication functionalities not to be significantly perturbed by the activation of checkpointing functionalities. The use of such a maximum value is recommended since it keeps the checkpoint latency at a minimum by keeping low the amount of bursts required for the checkpoint operation.

request at the application level. It is incremented by the control program each time an EBUS DMA data transfer (from/to host memory) associated with the checkpoint operation is completed. The value of this counter is used by the re-synchronization function `ckpt_cond_abort(float threshold)`, where the parameter `threshold` indicates the completion percentage of the last activated checkpoint operation, if any, under which the checkpoint operation must be aborted. To decide whether to abort the checkpoint operation or not, `ckpt_cond_abort()` needs information about both the current value of the `completed_transfers` counter and the total number of EBUS DMA data transfers required for the operation. The latter information is maintained into an additional variable located onto host memory, namely `total_transfers`, which is visible to `ckpt_cond_abort()` as well as to the function `non_block_ckpt()`. As soon as the checkpoint operation is issued by the application, the function `non_block_ckpt()` computes the total number of EBUS DMA transfers (from/to host memory) to complete the operation. The variable `total_transfers` stores the obtained result, making it available to the `ckpt_cond_abort()` function, which takes the decision on whether to abort or not the last activated checkpoint operation on the basis of the condition  $\frac{\text{completed\_transfers}}{\text{total\_transfers}} < \text{threshold}$ . The operation is aborted if the condition is verified. In such a case, the function `ckpt_cond_abort()` sets a flag located in the LANai internal memory, namely `ckpt_abort`, indicating to the control program run by the LANai processor that no additional EBUS DMA data transfer associated with checkpointing needs to be programmed for that operation. If previous condition is not verified, software at the host side spinlocks around a flag, namely `ckpt_complete`, also located in the LANai internal memory, thus resulting in temporary freezing of any simulation operation carried out by the CPU. The flag is reset by the control program as soon as the checkpoint operation is completed.

### 3. MINIMUM COST RE-SYNCHRONIZATION

In this section we present the checkpointing-recovery cost model and the  $\mathcal{MC}$  re-synchronization semantic relying on the model. Then we discuss how to solve the model and provide details on the re-synchronization function we have developed to implement  $\mathcal{MC}$ .

#### 3.1 The Cost Model and the $\mathcal{MC}$ Semantic

Let us consider a portion of the evolution of an LP in simulation time as shown in Figure 2. Black circles represent simulation events already executed by the LP, empty circles represent events not yet executed, and labeled boxes represent state vector values at given points in simulation time that is, those points corresponding to event timestamps. A state vector value is associated with each executed event. This value results from updates issued by the LP on state variables while executing that event. In our example, the state vector value  $X$  results from updates issued on the state vector value  $Y$  due to the execution of the event  $e$ . We associate with each state vector value  $X$  traversed by the LP a probability value, namely  $P(X)$ , which is the probability that a future rollback needs restoration to the state vector value  $X$ .

Suppose we are currently taking a snapshot of the state

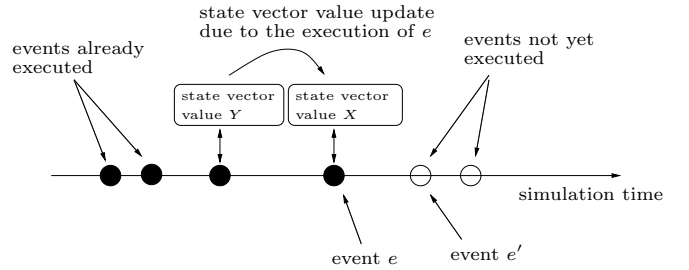


Figure 2: Evolution of the LP in simulation time.

vector value  $X$  using the non-blocking execution mode of the checkpointing protocol (i.e. the EBUS DMA is transferring data from the LP state vector into the checkpoint stack) and re-synchronization occurs either because the LP has been re-scheduled for the execution of its next event, namely  $e'$  in Figure 2, or because some other LP needs to issue a non-blocking checkpoint request. At this point we must establish the convenience of committing/aborting the non-blocking checkpoint operation currently involving the state vector value  $X$ . We recall that commit causes temporarily freezing of simulation software run at the host side until the completion of the checkpoint operation. Instead, abort leads to skipping that checkpoint, with possible impact on the recovery latency due to the reduced amount of available checkpoints.

Denoting with  $\Delta_{\text{completion}}$  the expected residual completion latency for the checkpoint operation at re-synchronization occurrence, the checkpointing overhead  $OH_{\text{ckpt}}$  associated with re-synchronization can be expressed as

$$OH_{\text{ckpt}} = \begin{cases} \Delta_{\text{completion}} & \text{commit case} \\ 0 & \text{abort case} \end{cases} \quad (1)$$

Actually,  $\Delta_{\text{completion}}$  corresponds to the freezing interval on CPU activities in case a commit decision is taken for the on-going checkpoint operation. Note that, while modeling  $OH_{\text{ckpt}}$ , we have approximated the CPU overhead for aborting the checkpoint as a null value. This approximation is justified by the fact that, as for the implementation of  $\mathcal{CCA}$ , abort is implemented in  $\mathcal{MC}$  by simply setting the value of the `ckpt_abort` flag (see Section 2), which notifies to the control program run by the LANai processor that EBUS DMA data transfers associated with the on-going checkpoint operation needs to be interrupted.

We know that taking the checkpoint of the state vector value  $X$  does not affect the recovery time to any state vector value  $Z$  preceding  $X$ . This is because the state recovery time to  $Z$  depends only on the position of the latest checkpoint preceding (or coinciding with)  $Z$ , and on the granularity (execution time) of the intermediate events, if any, from that checkpoint to the state vector value  $Z$ . Specifically, if  $Z$  has not been checkpointed and must be recovered due to rollback, then coasting-forward (i.e. event execution replay) is needed from the latest checkpoint preceding  $Z$ , to  $Z$ . On the other hand, if  $Z$  has been checkpointed, then state recovery only entails reloading the checkpointed value into the LP state buffer. In both cases, the time for the state recovery operation is not affected by the commit/abort of the checkpoint operation involving the state vector value  $X$ .

By the previous discussion, the recovery overhead asso-

ciated with re-synchronization, denoted as  $OH_{recovery}$ , can be evaluated as the additional expected recovery cost due to recovery of the state vector value  $X$  in case of rollback. Such a cost varies depending on whether the on-going checkpoint operation involving  $X$  is committed or aborted. Denoting with (i)  $EV(X)$  the set of all the events that move the LP from the latest checkpointed state vector value preceding  $X$ , to  $X$ , (ii)  $\Delta_e$  the granularity of the event  $e \in EV(X)$ , and (iii)  $\Delta_{reload}$  the time to reload a checkpointed state vector value into the state buffer of the LP, we can express  $OH_{recovery}$  as

$$OH_{recovery} = \begin{cases} P(X)\Delta_{reload} & \text{commit case} \\ P(X)[\Delta_{reload} + \sum_{e \in EV(X)} \Delta_e] & \text{abort case} \end{cases} \quad (2)$$

Expression (2) states that if the checkpoint operation involving  $X$  is eventually committed, then in case of state recovery to  $X$  (this happens with probability  $P(X)$ ), the recovery overhead consists only of the time  $\Delta_{reload}$  to reload the checkpointed state vector value  $X$  into the LP state buffer. Otherwise, it consists of the time to reload the latest checkpointed state vector value preceding  $X$  plus the time to replay all the events in  $EV(X)$  that is, the coasting-forward time. Summing contributions in (1) and (2) we get the following expression for the whole checkpointing-recovery overhead associated with re-synchronization occurrence

$$OH_{ckpt} + OH_{recovery} = \begin{cases} \Delta_{completion} + P(X)\Delta_{reload} \\ P(X)[\Delta_{reload} + \sum_{e \in EV(X)} \Delta_e] \end{cases} \quad (3)$$

We note that the expression for  $OH_{recovery}$  has been derived by implicitly assuming that  $P(X)$  does not change depending on whether the on-going checkpoint operation involving the state vector value  $X$  is committed or aborted. In other words, we have assumed that committing or aborting the checkpoint will result in no perceptible change in the rollback behavior. As respect to this point, we note that committing or aborting a checkpoint at a specific re-synchronization point will ultimately result in shorter or longer average distance between checkpoints for the LP. As commonly assumed by performance models for CPU charged checkpointing [15, 19, 21, 22, 25, 30, 31], such a phenomenon has in practice no significant effect on the rollback behavior.

The  $\mathcal{MC}$  re-synchronization semantic should commit/abort the on-going checkpoint operation in order to minimize the checkpointing-recovery overhead as expressed in (3). However, we note that in case the last activated non-blocking checkpoint operation has been already completed upon re-synchronization occurrence, then no abort decision makes sense. This is reflected by the checkpointing-recovery cost model since already completed checkpoint actually means a null value for  $\Delta_{completion}$ , which in turn leads the commit decision to always minimize the checkpointing-recovery overhead due to the fact that  $P(X)\Delta_{reload} \leq P(X)[\Delta_{reload} + \sum_{e \in EV(X)} \Delta_e]$ .

This observation allows us to formally define  $\mathcal{MC}$  in a way that sometimes we can avoid to evaluate the cost model in (3) upon re-synchronization, i.e. when the non-blocking checkpoint operation has been already completed. Denoting with  $x$  and  $y$  the values obtained by evaluating the cost

---

```

1. if last activated checkpoint operation already completed
2.   <return COMMIT>
3. <compute value = x - y >
4. if value ≤ 0
5.   <wait for checkpoint completion>
6.   <return COMMIT>
7. else
8.   <interrupt the on-going checkpoint operation>
9.   <return ABORT>

```

---

Figure 3: The algorithm for  $\mathcal{MC}$ .

model in (3) in case of commit and abort, respectively, our definition of  $\mathcal{MC}$  is reported in Figure 3. From a practical view point, the test in line 1 and the statement in line 2 allow a reduction of the overhead associated with  $\mathcal{MC}$  since, in case the test is verified and the statement is executed, no calculation of  $value = x - y$  must be performed (i.e. the cost model does not need to be solved upon re-synchronization).

### 3.2 Solving the Model

Solving the checkpointing-recovery cost model, i.e. computing the value  $x - y$  in line 3 of the algorithm in Figure 3, requires knowledge of several parameter values. Some of these parameters, namely the event execution time  $\Delta_e$ , and the probability  $P(X)$  of recovery to a given state vector value, appear in the literature in several performance models for classical CPU charged checkpointing, and a set of solutions have been already proposed for determining their values<sup>6</sup>. For space constraints we cannot survey these solutions, anyhow, the reader can refer to [8, 22, 25] for the treatment and determination of these parameters.

A single parameter appears in the cost model in expression (3), whose determination has not been already treated in the literature, since it does not appear in any already proposed performance model for CPU charged checkpointing. This is the expected residual completion latency of the non-blocking checkpoint operation, namely  $\Delta_{completion}$ , for which we present a determination method in what follows.

By the description in Section 2, we know that any non-blocking checkpoint operation is split by the control program into a sequence of EBUS DMA data transfer operations (bursts) from the host memory, i.e. from the LP state vector to the LANai internal memory, and vice versa, i.e. from the LANai internal memory to the stack of checkpoints located in the host memory. The counters `total.transfers` and `completed.transfers` maintained by the current implementation of CCL (see Section 2) indicate, respectively, the total amount of bursts required by the last activated non-blocking checkpoint operation and the amount of bursts which have already been completed. As a consequence, the difference between the two counter values indicates the amount of bursts that still need to be carried out in order to complete that checkpoint operation.

Denoting with  $\Delta_{burst}$  the expected time for a single burst performed by the EBUS DMA then we get the following

---

<sup>6</sup>For what concerns  $\Delta_{reload}$ , we note that computation of  $value = x - y$  in line 3 of the algorithm in Figure 3 actually does not really need knowledge of this parameter. In other words, solving the cost model as the difference between  $x$  and  $y$  allows us to ignore that parameter value.

lower bound  $L$  for  $\Delta_{completion}$

$$L = (total\ transfers - completed\ transfers)\Delta_{burst} \quad (4)$$

This lower bound is obtained in case no message needs to be transferred into the receive queue through the EBUS DMA during a period that starts just upon re-synchronization occurrence and ends with the completion of the checkpoint operation. We refer to this period as *re-synchronization period*. However, it is possible that messages incoming from the network need to be transferred into the receive queue through the EBUS DMA during the re-synchronization period. Given that message transfer has higher priority as compared to data transfer associated with checkpointing, the real value of  $\Delta_{completion}$  might be actually larger than the lower bound  $L$ . The real value of  $\Delta_{completion}$  actually corresponds to the re-synchronization period length, therefore, our objective is to evaluate the expected length of the re-synchronization period.

Denoting with  $M$  the number of messages already buffered into the LANai internal memory upon re-synchronization occurrence, which need to be transferred into the receive queue through the EBUS DMA, and with  $\Delta_{message}$  the expected time required by the EBUS DMA for transferring a single message from the LANai internal memory into the receive queue, we get that the expected length of the re-synchronization period, namely  $\Delta_{completion}$ , can be expressed as

$$\Delta_{completion} = L + M\Delta_{message} + L' \quad (5)$$

where  $L$  is the previously mentioned lower bound,  $M\Delta_{message}$  is the time for transferring those  $M$  messages into the receive queue through the EBUS DMA, and  $L'$  is the time required for transferring into the receive queue the additional messages incoming from the network during the re-synchronization period. Denoting with  $f$  the expected frequency of message arrival from the network during the re-synchronization period we get

$$L' = f\Delta_{completion}\Delta_{message} \quad (6)$$

where  $f\Delta_{completion}$  represents the amount of messages incoming from the network during the re-synchronization period.

Plugging (4) and (6) in (5), we get for  $\Delta_{completion}$  the following expression

$$\Delta_{completion} = \frac{1}{1 - f\Delta_{message}} [M\Delta_{message} + (total\ transfers - completed\ transfers)\Delta_{burst}] \quad (7)$$

By (7), calculation of  $\Delta_{completion}$  requires knowledge of  $M$ ,  $\Delta_{burst}$ ,  $\Delta_{message}$  and  $f$ . The current implementation of CCL already maintains a counter in the LANai internal memory (managed by the LANai control program and accessible by software run at the host side) keeping track at any instant of the amount of messages already buffered into the LANai internal memory, which need to be transferred into the receive queue. (The counter value is used by the LANai control program to manage the activation of block-DMA operations for transferring those messages.) Therefore, the value of  $M$  is straightforwardly available to the

re-synchronization function implementing  $\mathcal{MC}$  just through this counter value.

In case a commit decision is taken for an on-going checkpoint operation, the re-synchronization function must wait for the completion of the operation in a way to temporarily suspend any other simulation operation carried out by the CPU. As for the case of  $\mathcal{CCA}$ , the re-synchronization function we have developed to implement  $\mathcal{MC}$  spinlocks around the flag `ckpt_complete`, which is set to one by the control program as soon as the operation is completed. As a consequence, the values of  $\Delta_{burst}$  and  $\Delta_{message}$  must be evaluated considering the interference on EBUS DMA activities due to access of the host CPU to the PCI bridge and to the LBUS caused by read operations continuously issued on the flag `ckpt_complete` while waiting for the completion of the checkpoint operation. (All the other components on board of the card have lower access priority to the LBUS as compared to the EBUS DMA [17], therefore they do not produce interference on data transfer performed by the EBUS DMA).

As respect to this point, measures reported in [24] have shown that the time to transfer a data block from/to the host memory through the EBUS DMA while the host CPU spinlocks around a flag located in the LANai internal memory increases linearly with the data block size. As a consequence, once fixed the message size and the burst size,  $\Delta_{burst}$  and  $\Delta_{message}$  can be assumed as constant values any time we hypothesize that no other PCI peripherals (such as audio cards) are using the same PCI bus. Since this can be considered as a common scenario when dedicating a cluster to a specific parallel computing application,  $\Delta_{burst}$  and  $\Delta_{message}$  can be determined while installing CCL (for example through the benchmarking described in [24]) and made available to the re-synchronization function implementing  $\mathcal{MC}$ .

By previous arguments,  $f$  is the only parameter that depends on proper dynamics of the specific parallel execution, whose value really needs to be determined upon re-synchronization occurrence. As pointed out before,  $f$  is the expected frequency of message arrival during the re-synchronization period, however we need this value at the start of the period itself in order to solve the cost model, therefore a prediction mechanism for determining this value must be encompassed.

According to the commonly accepted belief that the recent past behavior should be a reliable indicator of the immediate future behavior (for instance this is the basic assumption underlying many dispatcher designs, where recent process/thread behavior, in terms of CPU bursts or I/O requests is assumed as representative of the immediate future behavior [29, 33]), we have decided to approximate  $f$  with the frequency of message arrival from the network in the interval between the activation of the last non-blocking checkpoint operation and the occurrence of re-synchronization. In other words, the prediction of the value of  $f$  is based on statistics related to the temporal window associated with the already executed portion of the checkpoint operation involved in the re-synchronization occurrence.

To compute that frequency value, we further exploit hardware features of the LANai 4 chip. Specifically, this chip is equipped with a Real Time Clock register (RTC), providing real time measures with granularity on the order of 0.5 microseconds, which can be reset/read by the host CPU. We have introduced an additional counter in the LANai in-

ternal memory, namely `incoming_messages`, which counts the amount of messages incoming from the network since the activation of the last non-blocking checkpoint operation. The counter is incremented by the control program run by the LANai processor each time it activates the Receive DMA on board of the card. Then we have slightly modified the function `non_block_ckpt()` in order to let it reset both `incoming_messages` and the register RTC upon the issue of a non-blocking checkpoint operation. The value of  $f$  to be used while computing  $\Delta_{completion}$  can be evaluated by the re-synchronization function implementing  $\mathcal{MC}$  as

$$f = \frac{\text{incoming messages}}{\$RTC} \quad (8)$$

As a last observation, we note that this estimation method for  $f$  is expected to produce in practice negligible overhead due to the fact that the time for resetting the values of RTC and `incoming_messages` is a negligible percentage of the whole execution time of the function `non_block_ckpt()`, despite reset itself needs access to the LANai internal memory passing through the PCI bridge. Specifically, upon its execution, the function `non_block_ckpt()` must provide the control program run by the LANai processor with any information required for programming the EBUS DMA for data transfer operations associated with checkpointing. This already happens through a sequence of accesses to the LANai internal memory in order to write that information into a proper buffer. Also, since the increment of the counter `incoming_messages` is implemented through few machine instructions, the corresponding overhead on the LANai control program is negligible in practice.

### 3.3 Details on the Re-synchronization Function Implementing $\mathcal{MC}$

The function developed to implement  $\mathcal{MC}$  has the prototype `min_cost_resynch(double prob, double cumulate)`, where `prob` corresponds to the probability value  $P(X)$  in the cost model in expression (3), and `cumulate` is the sum of the execution costs, expressed in microseconds, of all the events executed by the LP since the last committed checkpoint operation (also this quantity appears in the cost model). In practice we have left all the classical parameters of the cost model (i.e. those appearing in the literature in performance models for CPU charged checkpointing) as parameters to be passed to `min_cost_resynch()`. Instead, the parameter  $\Delta_{completion}$  is handled by CCL in a transparent way to the application programmer, according to the solution previously described. This is an engineering choice allowing the application level programmer to use the preferred solution, for example one among those proposed in [8, 22, 25], to tackle the determination problem of the classical parameters. As already pointed out, the value of the parameter  $\Delta_{reload}$  is not required for solving the cost model as the difference between  $x$  and  $y$  (see line 3 of the algorithm in Figure 3). This is the reason why we have introduced no parameter for passing the value of  $\Delta_{reload}$  to the function `min_cost_resynch()`.

## 4. EXPERIMENTAL RESULTS

In [27], results of an extended performance study of non-blocking checkpointing with the  $\mathcal{MC}$  re-synchronization semantic are reported. The study considers both a wide set

of classical simulation benchmarks, derived from the well known PHOLD model [12], and mobile systems simulation applications. For space constraint we cannot report that complete set of data, therefore we present in this section data related to optimistic parallel simulation of a specific Personal Communication System (PCS) model, selected as a representative real world application among those employed in [27]. As a testing environment for this application we have used a cluster of 8 Pentium II 300 MHz (128 Mbytes RAM - 512 Kbytes second level cache). All the PCs of the cluster run LINUX (kernel version 2.0.32) and are equipped with M2M-PCI32C myrinet cards.

The simulation software we have used implements the events as a compound structure with several fields (sender, receiver, timestamp etc.), having total size of 60 bytes. Using CCL any message carrying an event is delivered to the recipient within about 20 microseconds when no congestion occurs on the myrinet switch. Such a delivery delay is aligned with delays provided for similar message sizes by other fast speed messaging layers for myrinet [18, 32]. Message exchange among LPs hosted by the same machine does not involve operations of the CCL layer. Each LP is implemented as an application-level thread and there is an instance of an optimistic simulation engine on each machine. The engine manages the local event list (resulting as the logical collection of the event lists of the local LPs) and schedules LPs for event execution according to the standard Smallest-Timestamp-First algorithm [16]. Dynamic memory allocation/release based on standard `malloc()/free()` calls is adopted for the entries of the event lists.

We do not limit our analysis to a pure comparison between the two re-synchronization semantics  $\mathcal{MC}$  and  $\mathcal{CCA}$  in support of non-blocking checkpointing, since we also report results related to CPU charged checkpointing in combination with a classical Periodic State Saving (PSS) approach that makes the LP to take a checkpoint each  $\chi$  event executions [9, 21, 25], where  $\chi$  is commonly referred to as checkpoint interval (<sup>7</sup>). CPU charged checkpointing is implemented efficiently through a `memcpy()` call. Furthermore, as CCL adopts reserved main memory pages for both the LPs state buffers and their stacks, to ensure fairness in the comparison we have used reserved main memory pages also for CPU charged checkpointing.

For the tests with non-blocking checkpointing, 1 Kbyte has been selected as the burst length for EBUS DMA operations on the basis of measurements reported in [24]. Also, a non-blocking checkpoint operation is activated by the LP after the execution of each simulation event, thus obtaining a situation in which lack of checkpoints for particular LP state vector values is determined only on the basis of checkpoint abort decisions taken upon re-synchronization. As shown by experimental results in [23], this is a favorable test case for non-blocking checkpointing, allowing to fully exploit the effectiveness of an optimized re-synchronization semantic. As a last preliminary observation, the expected event granular-

<sup>7</sup>Although for some particular simulation problems (e.g. simulations with very regular patterns in the difference between timestamps of consecutive events) performance improvements over PSS can be achieved by relaxing the constraint that CPU charged checkpoints must be taken on a periodic basis [22], PSS remains, in general, a performance effective solution. This is the reason why we take it as a reference point in the experimental study.

ity for the used PCS model is known in advance, therefore we have used such an expected value as the execution cost  $\Delta_e$  of any event in the set  $EV(X)$  of the cost model (see expression 3). Also, the parameter  $P(X)$ , namely the probability of rollback to a given state  $X$ , has been simply estimated as the rollback frequency (i.e. the ratio between the number of rollbacks and the number of executed events) of the LP in order to favor simplicity and avoid the overhead of more complex estimation methods [25].

A PCS provides communication services to mobile units. In our simulation model the service area is partitioned into cells, each of which is modeled by a distinct LP. Each cell represents a receiver/transmitter having either a fixed number of channels allocated to it (Fixed-Channel-Assignment, namely FCA) or a number of channels dynamically assigned to it (Dynamic-Channel-Assignment, namely DCA). In this paper we consider an FCA model with 100 channels per cell. The model is call-initiated [6] 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). The average holding time for each call is 2 minutes, and call requests arrive to each cell according to an exponential distribution [1, 4, 6] with inter-arrival time 1.5 seconds, thus originating channel utilization factor of about 80%.

There are three types of events, namely hand-off (due to mobile unit cell switch), call termination and call arrival. When a call arrives at a cell, channel availability is determined by exploiting information maintained into a list of available channels. If all channels are busy, the incoming call is simply counted as a “block”. If at least one channel is available, channel assignment for the new call takes place and the list of available channels is updated accordingly. A call termination simply involves the release of the associated channel and statistics update. Hand-off takes place each time a mobile unit currently involved in conversation moves from one cell to another. 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, with mean 3 minutes (fast movement units) and 30 minutes (slow movement units), respectively. 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. 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. Instead, the hand-off event at the destination cell checks for channel availability. If there is no available channel, then the call is simply cut off (dropped), otherwise an available channel is assigned to the call. Hand-off events for destination cells are not pre-computed, i.e. they are scheduled only upon the occurrence of the hand-off event at the cell left by the mobile unit.

The state vector of any LP records statistics, information about busy channels and, for each channel, information about features of the mobile unit involved in the on-going call (e.g. scheduled call termination time, call initiation time, class of the mobile unit etc.), if any. As a result, the size of the state vector is about 4 Kbytes<sup>(8)</sup>. The ex-

<sup>8</sup>PCS models might exhibit smaller LP state granularity [4, 6]. This might happen, for example, when information maintained within the LP state vector is almost exclusively re-

lated to the amount of busy channels within the associated cell.

pected event granularity for this model, including the cost of statistics update, is on the order of 35 microseconds on the adopted architecture.

We have simulated a PCS in which each cell is hexagonal, therefore all the cells, except bordering cells of the coverage area, have six neighbor cells. The model size, in terms of number of cells, i.e. number of LPs, has been varied between 32 and 256. The LPs have been evenly distributed among the 8 machines of the cluster. Actually, variation of the model size while maintaining the same size for the underlying architecture allows us to provide results for different degrees of parallelism in the simulation model execution, which typically originate different amounts of rollback in the parallel execution.

We report results related to the *event rate*, classically evaluated as the number of committed simulation events per second. This parameter indicates how fast is the parallel execution with a given checkpointing/re-synchronization scheme. For the case of non-blocking checkpointing with *CCA*, we report the peak event rate observed while moving the parameter **threshold** from 0.0 to 1.0, with step 0.1. Similarly, for *PSS* we report the peak event rate observed vs  $\chi$ . The checkpoint interval  $\chi$  and the **threshold** value originating those peak event rate values are also reported. We additionally report results related to the *efficiency* of the parallel execution, classically evaluated as the ratio between the amount of committed simulation events and the total number of executed events (committed plus rolled back). This parameter is an indicator of the percentage of productive simulation work performed since it is a measure of the probability for an event not to be eventually rolled back once executed. For the *MC* re-synchronization semantic we also report the speedup achieved by the parallel execution, namely the ratio between parallel execution time on the 8 machines and serial execution time of the same simulation model on a single machine of the cluster. By observing this measure we want to ascertain whether performance results reported are obtained in a condition where parallel simulation is really effective. Mean-Value-Analysis has been adopted while collecting statistics and each reported value is the average of 20 runs, all done with different seeds for the pseudo-random generators. At least  $2 \times 10^6$  committed simulation events have been executed in each run. The results are displayed in Figure 4.

The plots for the event rate show a clear gain in the execution speed obtained by non-blocking checkpointing through *MC*. Such a gain is up to 7% as compared to the event rate achieved with *CCA* and, more important, it is up to 13% as compared to the event rate of *PSS*. Higher gains are observed in case of smaller size of the simulation model, which originates a higher degree of parallelism in the model execution. This is an expected behavior when looking at the efficiency plots. Specifically, when the degree of parallelism in the model execution gets lower, i.e. the model size is increased, we obtain efficiency values in the order of up to 95%, denoting a very low amount of rollback in the parallel execution (events are less likely to be rolled back once executed). In this situation, any optimized checkpointing approach, whether it be based on CPU charged or DMA based checkpointing, typically allow to almost eliminate the checkpointing overhead while paying in practice negligible

lated to the amount of busy channels within the associated cell.

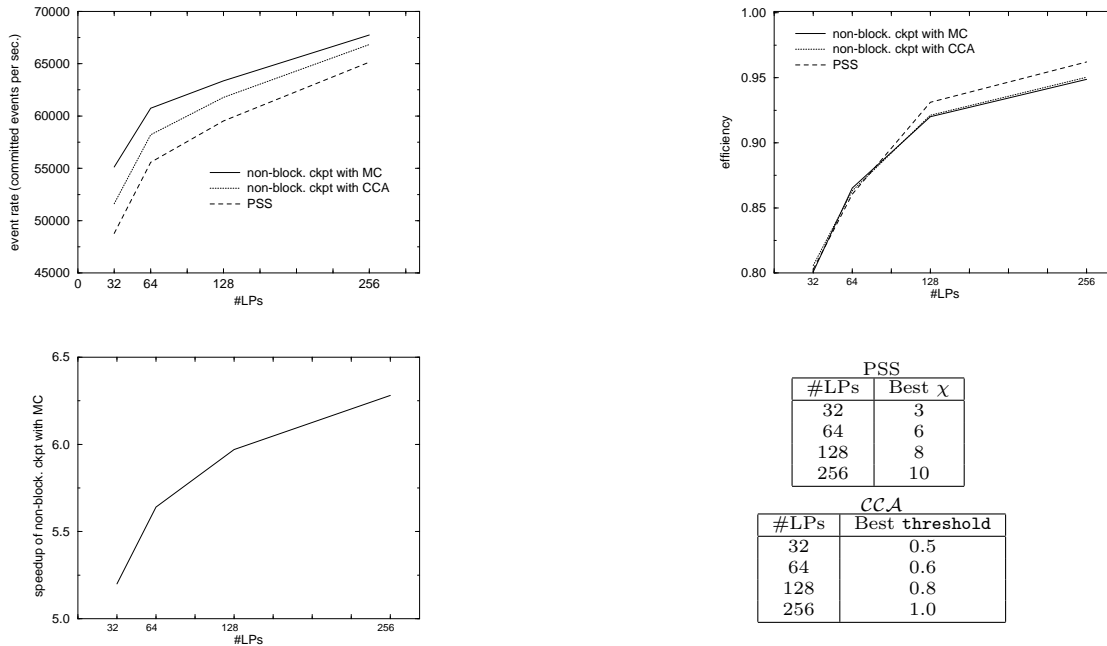


Figure 4: Results for the PCS simulation application.

coasting-forward costs due to the fact that rollbacks are infrequent. As respect to this point, in case of model size 256, PSS achieves the best performance for a large value of the checkpoint interval  $\chi$ , which just indicates the strong reduction of the checkpointing overhead achieved even in case of CPU charged checkpoints at the point in which the event rate is maximized vs  $\chi$ . A similar observation can be drawn for non-blocking checkpointing with *CCA* when considering that, for model size 256, the best **threshold** value is 1.0, which means that any on-going checkpoint is actually aborted upon re-synchronization. On the other hand, when the amount of rollback is non-minimal, i.e. for efficiency values in the order of 85% or less, non-blocking checkpointing with *MC* allows significant performance improvements, which shows that *MC* reveals a more effective solution for generic rollback patterns.

Finally, we note that the speedup achieved by non-blocking checkpointing with *MC* ranges between 65% and 78% of the ideal speedup achievable on 8 machines, which is larger than the commonly recognized threshold-speedup, namely 50% of the ideal one, determining reasonable accelerations in case of parallel simulation on distributed memory systems.

## 5. SUMMARY

In this paper we have presented a performance model for non-blocking, i.e. DMA based, checkpointing in support of optimistic parallel discrete event simulation on myrinet clusters. By the model, we define a performance effective re-synchronization semantic, namely minimum cost, between CPU activities and checkpointing activities carried out through DMA.

We have implemented the minimum cost semantic within the Checkpointing-and-Communication Library (CCL), already supporting the non-blocking execution mode of the

checkpointing protocol. We have also reported data related to the benefits from minimum cost re-synchronization, in terms of execution speed of the parallel simulation application, for the case of a personal communication system simulation model.

## 6. REFERENCES

- [1] A. Boukerche, S. K. Das, A. Fabbri, and O. Yildiz. Exploiting model independence for parallel PCS network simulation. In *Proc. of the 13th Workshop on Parallel and Distributed Simulation*, pages 166–173. ACM/IEEE Computer Society, May 1999.
- [2] J. Briner. Fast parallel simulation of digital systems. In *Proc. of Multiconf. on Advances in Parallel and Distributed Simulation*, pages 71–77, 1991.
- [3] D. Bruce. The treatment of state in optimistic systems. In *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, pages 40–49. ACM/SCS, June 1995.
- [4] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: a high performance modular Time Warp system. In *Proc. of the 14th Workshop on Parallel and Distributed Simulation*, pages 53–60. ACM/IEEE Computer Society, May 2000.
- [5] C. D. Carothers, R. M. Fujimoto, P. England, and Y. B. Lin. Distributed simulation of large-scale PCS networks. In *Proc. of the 2nd IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 2–6. IEEE Computer Society, 1994.
- [6] C. D. Carothers, R. M. Fujimoto, and Y. B. Lin. A case study in simulating PCS networks using Time Warp. In *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, pages 87–94. ACM/SCS, June

- 1995.
- [7] E. Elnozahy, D. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. of the 11th Symposium on Reliable Distributed Systems*, pages 39–47. IEEE Computer Society, Oct. 1992.
  - [8] A. Ferscha and J. Luthi. Estimating rollback overhead for optimism control in Time Warp. In *Proc. of the 28th Annual Simulation Symposium*, pages 2–12. IEEE Computer Society, Apr. 1995.
  - [9] J. Fleischmann and P. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. In *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58. ACM/SCS, June 1995.
  - [10] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Trans. of the Society for Computer Simulation*, 6(3):211–239, 1989.
  - [11] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.
  - [12] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proc. of the Multiconf. on Distributed Simulation*, pages 23–28. Society for Computer Simulation, Jan. 1990.
  - [13] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and System*, 7(3):404–425, July 1985.
  - [14] K. Li, J. Naughton, and J. Plank. Low latency concurrent checkpointing for parallel programs. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):474–479, Aug. 1994.
  - [15] Y. Lin, B. Preiss, W. Loucks, and E. Lazowska. Selecting the checkpoint interval in Time Warp simulation. In *Proc. of the 7th Workshop on Parallel and Distributed Simulation*, pages 3–10. ACM/SCS, 1993.
  - [16] Y. B. Lin and E. D. Lazowska. Processor scheduling for Time Warp parallel simulation. In *Advances in Parallel and Distributed Simulation*, pages 11–14, 1991.
  - [17] MYRICOM. LANai 4. *Draft*, Feb. 1999.
  - [18] S. Pakin, M. Lauria, and A. Chen. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. of Supercomputing'95*. ACM/IEEE Computer Society, Dec. 1995.
  - [19] A. C. Palaniswamy and P. A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 127–134. ACM/SCS, 1993.
  - [20] J. Plank, M. Beck, and G. Kingsley. Libckpt: Transparent checkpointing under UNIX. In *Proc. of USENIX Winter Technical Conference*, pages 213–223. USENIX Association, 1995.
  - [21] B. R. Preiss, W. M. Loucks, and D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Trans. on Modeling and Computer Simulation*, 4(3):223–253, July 1994.
  - [22] F. Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Trans. on Parallel and Distributed Systems*, 12(4):346–362, Feb. 2001.
  - [23] F. Quaglia and A. Santoro. Nonblocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Trans. on Parallel and Distributed Systems*, 14(6):593–610, June 2003.
  - [24] F. Quaglia, A. Santoro, and B. Ciciani. Tuning of the checkpointing and communication library for optimistic simulation on Myrinet based NOWs. In *Proc. of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 241–248. IEEE Computer Society, Oct. 2001.
  - [25] R. Ronngren and R. Ayani. Adaptive checkpointing in Time Warp. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation*, pages 110–117. ACM/SCS, July 1994.
  - [26] R. Ronngren, M. Liljenstam, R. Ayani, and J. Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proc. of the 10th Workshop on Parallel and Distributed Simulation*, pages 70–77. ACM/IEEE Computer Society, May 1996.
  - [27] A. Santoro. *Semi-Asynchronous Checkpointing for Optimistic Parallel Simulation*. PhD thesis, Dipartimento di Informatica e Sistemistica, University of Rome La Sapienza, Feb. 2003.
  - [28] A. Santoro and F. Quaglia. Benefits from semi-asynchronous checkpointing for Time Warp simulations of a large state PCS model. In *Proc. of the Winter Simulation Conference*, pages 1339–1345. Society for Computer Simulation, Dec. 2001.
  - [29] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.
  - [30] S. Skold and R. Ronngren. Event sensitive state saving in Time Warp parallel discrete event simulation. In *Proc. of the Winter Simulation Conference*, pages 653–660. Society for Computer Simulation, Dec. 1996.
  - [31] H. Soliman and A. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Trans. on Parallel and Distributed Systems*, 9(10):947–951, Oct. 1998.
  - [32] S. Srinivasan, M. J. Lyell, P. F. Reynolds, Jr., and J. Wehrwein. Implementation of reductions in support of PDES on a network of workstations. In *Proc. of the 12th Workshop on Parallel and Distributed Simulation*, pages 116–123. ACM/IEEE Computer Society, May 1998.
  - [33] W. Stallings. *Operating Systems, Internals and Design Principles*. Prentice Hall, 1998.
  - [34] J. Steinman. Incremental state saving in SPEEDES using C plus plus. In *Proc. of the Winter Simulation Conference*, pages 687–696. Society for Computer Simulation, Dec. 1993.
  - [35] D. West and K. Panesar. Automatic incremental state saving. In *Proc. of the 10th Workshop on Parallel and Distributed Simulation*, pages 78–85. ACM/IEEE Computer Society, May 1996.
  - [36] F. Wieland. Practical parallel simulation applied to aviation control. In *Proc. of the 15th Workshop on Parallel and Distributed Simulation*, pages 109–116. ACM/IEEE Computer Society, May 2001.