ELSEVIER

Parallel Computing 27 (2001) 1227–1252

PARALLEL COMPUTING

www.elsevier.com/locate/parco

A checkpointing–recovery scheme for Time Warp parallel simulation

Vittorio Cortellessa [a,*], Francesco Quaglia [b]

[a] *Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Torvergata", Via di Tor Vergata, 00133 Roma, Italy*

[b] *Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy*

Received 28 March 1999; received in revised form 1 August 2000; accepted 29 November 2000

Abstract

This paper presents a checkpointing–recovery scheme for Time Warp parallel simulation. The scheme relies on a checkpointing protocol, namely mixed state saving, embedding both sparse and incremental state saving modes, and on a state recovery procedure embedding both forward and backward recovery modes. This scheme is a generalization of many previous solutions, which can be obtained as particular instances of it by selecting appropriate values for the checkpointing protocol parameters. We also present two regulating algorithms to adaptively tune the checkpointing protocol parameters, in order to make the protocol reacting to variable rollback behavior. A synthetic benchmark in several different configurations has been used for evaluating and comparing our scheme with previous solutions. The obtained data show that our solution allows faster execution and, in addition, keeps quite low the amount of memory used for recording state information; this allows the scheme to not adversely affect performance when memory is a critical resource. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Optimistic synchronization; Time Warp; Rollback-recovery; Performance optimization; Parallel discrete event simulation

1. Introduction

In parallel discrete event simulation, the simulator is partitioned into a set of logical processes (LPs), each one modeling a distinct part of the simulated system.

[*] Corresponding author.
E-mail addresses: cortelle@info.uniroma2.it (V. Cortellessa), quaglia@dis.uniroma1.it (F. Quaglia).

The execution of a simulation event at an LP (i) possibly moves the LP from one state to another, (ii) moves the simulation clock, namely local virtual time (LVT), of the LP to the event timestamp and (iii) possibly produces new events to be executed by any LP. After it is produced, an event is sent to the destination LP via a message. To ensure correct simulation results, a synchronization mechanism must be used to guarantee timestamp ordered execution of events at each LP.

One of the most adopted synchronization mechanisms is Time Warp [14], that follows an optimistic approach, thus relying on no "block until safe" policy for event execution. In this mechanism any LP is allowed to execute events unless its pending event set is empty. Anytime the LP detects it has processed events out of timestamp order (i.e., the LP receives a message carrying an event with timestamp lower than its LVT), a rollback procedure recovers the LP state to an error free value. The main advantage of this type of synchronization, compared to blocking based synchronization mechanisms [6], is that it offers the potential for greater exploitation of parallelism. Empirical evidence has shown that Time Warp has allowed good speedup for simulations of combat models [33], queuing networks [10], communication networks [5,18], logic circuits [3] and others.

One of the core problems of the Time Warp synchronization mechanism is how to support fast state recovery with low checkpointing overhead. To tackle this problem a number of checkpointing–recovery schemes have been proposed, which exhibit different tradeoffs between the checkpointing overhead, evaluated in terms of both CPU time and memory usage, and the state recovery latency [2,8,9,15–17,19–21,23,27]. Some of these schemes [8,17,20,23] also cope with variable rollback behavior that can be originated by a number of causes such as variations of the load on the processors and possible phase behavior of the LPs in the lifetime of the simulation.

In this paper, we present a checkpointing–recovery scheme consisting of: (i) a checkpointing protocol, namely *mixed state saving*, that is a modification of a protocol presented in [9], and (ii) an associated recovery procedure whose peculiarity is the embedding of both classical backward and forward recovery modes. Let us stress that no existing recovery procedure adopts both these modes in conjunction. We also perform an analysis of the checkpointing overhead and recovery latency vs the checkpointing protocol parameters and present two regulating algorithms for tuning adaptively these parameters. These algorithms allow the scheme to cope with the possibly variable behavior of the overlying application. In addition, we briefly discuss implementation issues to support the scheme.

We show also how our scheme is actually a generalization of many previous solutions, which can be obtained as instances of it when considering particular values for the checkpointing protocol parameters.

Performance data for a case study on a classical benchmark, namely the PHOLD model [11], in several different configurations are reported for a comparison with previous solutions. The data show that our scheme actually leads to good performance and low memory usage in every case tested.

The remainder of the paper is organized as follows. Section 2 reports a detailed description of the Time Warp mechanism for parallel discrete event simulation. In Section 3 an overview of previous checkpointing–recovery schemes is provided. Our solution is presented in Section 4. Section 5 contains the analysis of the checkpointing overhead and recovery latency vs the checkpointing protocol parameters, and the tuning algorithms for these parameters. Implementation issues are discussed in Section 6. Performance data are reported in Section 7.

2. The Time Warp mechanism

In the Time Warp mechanism each LP has its own notion of simulation time, namely LVT. LPs interact solely by exchanging messages and any message exchange represents the scheduling of an event for the recipient LP. Each message carries the content of the scheduled event and is "stamped" with a *virtual send time* and a *virtual receive time*. Virtual send time is the LVT of the sender LP at the time the message is sent. Virtual receive time, also known as timestamp, represents the simulation time for the occurrence of that event at the recipient LP. Any event execution moves the LVT of the LP to the event timestamp, possibly moves the LP from one state to another (i.e., it possibly updates the value of some or all the state variables) and possibly produces new events to be executed by any LP.

Incoming events are stored by the LP into an event-queue. The event-queue is logically partitioned into *future-event-queue* and *past-event-queue*. The future-event-queue stores events received but not yet executed, instead the past-event-queue records already executed events.

No "block until safe" policy is considered for event execution, therefore any LP is allowed to execute pending events unless its future-event-queue is empty. This allows the possibility of timestamp order violation as an LP may receive a message carrying an event with timestamp lower than its LVT. If a timestamp order violation is detected, all the events that were executed out of order are rolled back (they are moved from the past-event-queue to the future-event-queue). Upon a rollback, the effects resulting from the execution of the events that are rolled back and involving other LPs must be undone. This is achieved by sending an antimessage for each event sent out during the rolled back portion of the simulation. [1] Upon the receipt of an antimessage associated with an already executed event, the recipient LP rolls back as well. If the event has not yet been executed, the antimessage has the only effect to "annihilate" the event. [2]

Rollback entails, therefore, (i) sending antimessages and (ii) recovering the LP state to its value prior to the timestamp order violation. Latter action pushes the LVT of the LP back to the timestamp of the last event executed in correct order. Point (i) requires that anytime an event is produced and sent out, the corresponding antimessage is produced as well and locally recorded for possible future sending.

[1] Actually an antimessage is an exact copy of the corresponding message, but with a negative sign.

[2] We recall that the Time Warp mechanism does not require communication channels between two LPs to be FIFO, therefore an antimessage may be delivered before the corresponding message arrives. In this case, the antimessage is simply placed into a queue of pending antimessages. Then, upon the receipt of the corresponding message, both of them are annihilated.

Point (ii) needs a checkpointing–recovery scheme; existing schemes are described in detail in Section 3.

The global virtual time, namely GVT, of a Time Warp simulation is defined as the minimum among the timestamps of events not yet executed, or currently being executed or carried on messages/antimessages still in transit. Therefore, GVT represents the commitment horizon of the simulation (i.e., all the events with timestamp less than GVT can never be rolled back, thereby they are committed events). The GVT notion is used both to recover memory allocated for data (events in the past-event-queue, antimessages and state information) which are obsolete and to allow operations that cannot be undone (e.g., interactions with external entities that cannot be required to rollback).

3. Related work

This section is devoted to a description of existing checkpointing–recovery schemes. Each scheme is a combination of a checkpointing protocol and an associated state recovery procedure.

The scheme originally proposed for Time Warp simulation relies on a checkpointing protocol known as *copy state saving* [14]. In this protocol, the state of an LP is saved into a *state queue* before every new event is executed. The saved state is marked with the current LVT of the LP. The associated recovery procedure is quite simple, as any past state is available for recovery purposes. If a timestamp order violation is detected at simulation time t, the recovery procedure simply entails retrieving the latest saved state with simulation time smaller than t and reloading this state into the current state buffer. This scheme adds to each event execution a checkpointing overhead, which is quantified by the sum of the time required to allocate a state buffer and the time required to copy the current state of the LP into the buffer. Good performance is obtained in case of simulations with LPs having small state granularity. Solutions to accelerate the checkpointing protocol through the usage of special hardware have been presented in [13]. They can provide high performance improvements, especially in the case of simulations with large state granularity, but are unattractive due to high cost and low portability. Finally, the memory usage of this scheme may reach unacceptable levels, unless a GVT calculation procedure, coupled with a memory recovering one, is frequently executed. Note, however, that frequent GVT calculation may have negative impact on performance.

Another widely used checkpointing–recovery scheme relies on a checkpointing protocol known as *sparse state saving* [12,15]. Compared to copy state saving, this protocol performs state saving less frequently, thus reducing both the CPU time spent saving state information and the memory usage. The drawback incurred is that the associated recovery procedure is more complex, as a state which must be restored may not be available. In this case, it must be recomputed by starting from a previous saved one and replaying intermediate events in a *coasting forward*, which adds to recovery a time penalty. Several (adaptive) policies for selecting the states to be saved have been proposed in order to achieve good tradeoffs between the time spent for checkpointing and the recovery latency [2,8,15–17,19–21,23,27].

A rather different scheme relies on a checkpointing protocol known as *incremental state saving* [1,29,31]. Compared to copy state saving, this protocol reduces both the checkpointing overhead and the memory usage. It maintains a history of before-images of the state variables that are modified during event execution. [3] Each before-image is recorded prior the state variable is updated. The associated state recovery procedure entails backwards re-traversing the logged history and copying before-images into their original state locations until the state to be restored is obtained. A main problem related to the incremental approach is the need for identifying the state variables that are updated by an event execution. Interesting efforts to achieve transparency through persistent objects [4] and overloaded C++ operators [24] have been proposed. A technique in which saving calls of before-images are automatically inserted by directly editing the application executable is presented in [32].

Another checkpointing–recovery scheme has been recently proposed, which relies on a checkpointing protocol, namely *multiplexed state saving* [9], combining features of previous protocols. In this protocol, before-images of the state variables are saved during the execution of each event; furthermore, the whole state of the LP is periodically saved each χ event executions. State recovery at simulation time t is accomplished by reloading the earliest saved state with time greater than t, if any, and applying before-images to the state variables. An analysis of this scheme to determine the best suited value of the interval χ has been recently presented in [26].

The checkpointing–recovery scheme we present in this paper shows some similarity with latter scheme for what concerns the checkpointing protocol. In particular, the checkpointing protocol we propose, namely *mixed state saving*, is a generalization of multiplexed state saving which uses saving calls of before-images only for a subset of the events, thereby allowing a reduction of the checkpointing overhead. Instead, the recovery procedure we propose, being a combination of coasting forward and before-images' application, is rather different from that of any previous scheme.

4. The scheme

We proceed in this section along the following steps: a discussion is pointed out on issues related to the recovery latency of previous schemes; then our scheme is presented as a solution for bounding that latency without incurring relevant additional checkpointing overhead.

4.1. Issues on the recovery latency in previous schemes

Let us consider the portion of the execution of LP_i shown in Fig. 1. As we are interested only in state transitions, log of state information, and state recovery

[3] The term "before-image" indicates the value of the state variable prior modifications due to the execution of an event.

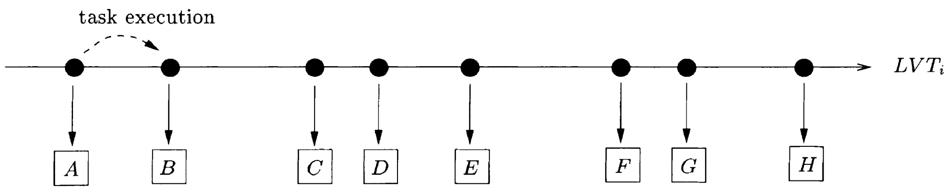

Fig. 1. A portion of the execution of LP_i.

actions, no detail about event-queues and communication between LPs is reported. The arrow extending toward the right-end represents simulation time, black circles represent timestamp values assumed by the LVT of LP_i, namely LVT_i, and labeled boxes represent state values, each one corresponding to a distinct value of LVT_i. State transitions and updates of LVT_i are determined by the execution of tasks, namely simulation events.

In what follows we evaluate the worst case recovery latency of previous checkpointing–recovery schemes as a function of the checkpointing protocol parameters. We do not explicitly include the scheme based on copy state saving in the discussion as it may be considered as a particular instance of the scheme based on sparse state saving when considering a fixed distance (in terms of executed events) between successive state saving operations equal to one.

Let us consider the situation shown in Fig. 2, where LP_i adopts sparse state saving. The saved states and the current state are explicitly shown in the figure. Considering the portion of the execution shown in the picture, the worst case for the recovery latency arises when LP_i has to roll back to the state labeled E, which immediately precedes the saved state labeled F. In this case, the recovery procedure entails restoring the state labeled A and replaying four intermediate events in order to update the state up to the value E (rollback to any state other than E would entail replaying less than four events). Previous example points out that the worst case for the recovery latency arises whenever LP_i has to rollback to a state which immediately precedes a saved one; in this case all the events, but the last one, executed between successive state saving operations must be replayed. Therefore, denoting with χ the average distance, in terms of executed events, between successive state saving operations, the worst case recovery latency is proportional to the quantity $\chi - 1$. Obviously, this latency can be kept short by imposing $\chi = 1$ (in this case we get that the checkpointing protocol boils down to copy state saving) at the expense of possibly unacceptable checkpointing overhead and memory usage.

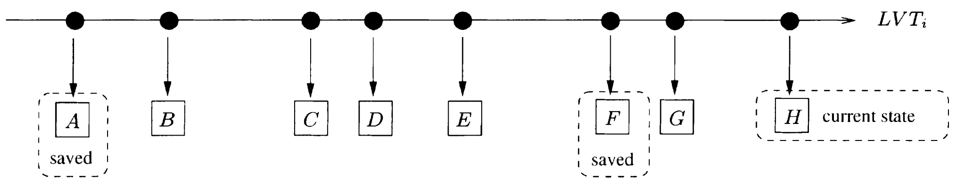

Fig. 2. Sparse state saving.

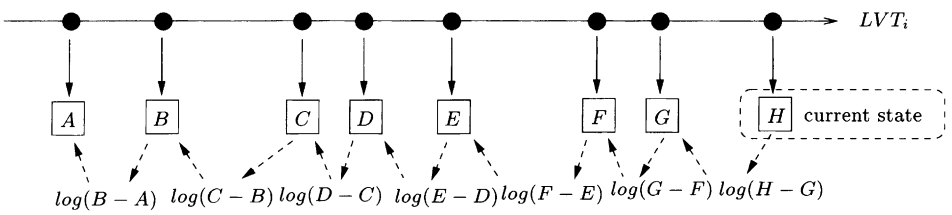

Fig. 3. Incremental state saving.

Let us now consider the case of the scheme relying on incremental state saving, shown in Fig. 3. The state saving protocol maintains logs of before-images of the state variables modified by event execution. The set of logged before-images of the state variables modified by an event moving the state of LP$_i$ from X to Y is denoted as $\log(Y - X)$. Let us suppose LP$_i$ must roll back to the state labeled B. In this case, six event executions must be undone and state recovery is accomplished by applying to the current state variables six logs of before-images. In the general case, the recovery latency is proportional to the extent of rollback (i.e., the number of undone event executions) and is *not otherwise controllable*. The absence in the checkpointing protocol of a tunable parameter allowing the recovery latency to be independent of the extent of rollback is a highly undesirable feature for the following main reason. The rollback extent strongly depends on the way the LVTs of the LPs diverge, which is related to several parameters, such as features proper of the application [4] and the load on the processors, just to name some. As a consequence, the average value of the rollback extent is unpredictable, potentially making the recovery latency unbounded [9]. The drawback incurred is that one cannot be sure prior to the execution whether the recovery latency of this scheme is a dominant factor adversely affecting performance or not.

The case of multiplexed state saving is shown in Fig. 4. Before-images are logged at each event execution and, in addition, the whole state is saved each $\chi = 5$ event executions. The worst case for the recovery latency arises when LP$_i$ must roll back to the state labeled B, which immediately follows the saved state labeled A. In this case, the recovery procedure entails restoring the saved state labeled F and applying four logs of before-images to the state variables (rollback to any state other than B requires to apply less than four logs). From previous example it is easy to deduce that the worst case for the recovery latency arises whenever the LP has to rollback to a non-saved state which immediately follows a saved one, as all the logs of before-images between successive state saving operations must be applied. Thereby, the worst case recovery latency is proportional to the quantity $\chi - 1$. As for the case of the scheme relying on sparse state saving, the latency can be kept short by imposing small values for χ, at the expense of possibly unacceptable checkpointing overhead and memory usage.

[4] As an example, how frequently LPs produce events for each other.

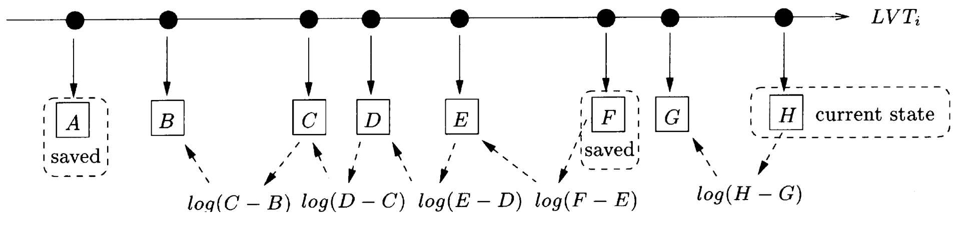

Fig. 4. Multiplexed state saving.

4.2. Mixed state saving and the associated recovery procedure

In this section, we present mixed state saving and the associated recovery procedure. Then issues related to the worst case recovery latency as a function of the checkpointing protocol parameters are pointed out.

Basically, mixed state saving is a modification of multiplexed state saving which allows lower checkpointing overhead and memory usage. It is derived from multiplexed state saving by avoiding to log before-images during the execution of some events. The protocol can be easily explained by looking at Fig. 5, where the same portion of the execution of LP_i considered in Section 3 is shown, and the same value for χ as that of the case of multiplexed state saving (i.e., $\chi = 5$) is considered. We underline that the case shown is a specific example since a same sequence of state saving actions happens each χ state transitions and the state H, which is the starting point of the recovery, can be any state in an interval.

States labeled A and F are saved. In addition, for a subset of the events which move the state of LP_i from A to F, the before-images of the state variables are logged. The events for which before-images are logged are selected as follows. After a state is saved, LP_i executes a number $\mu \leqslant \chi - 1$ of events without saving before-images, then for the remaining $\nu = \chi - 1 - \mu$ events executed prior to the next state saving operation, before-images of the state variables are logged. For the example shown in Fig. 5, we have $\mu = 2$, $\nu = 2$ and the set of logs of before-images is $\{\log(E - D), \log(F - E)\}$. As only some before-images are logged while moving from A to F, the checkpointing protocol introduces an overhead which is smaller

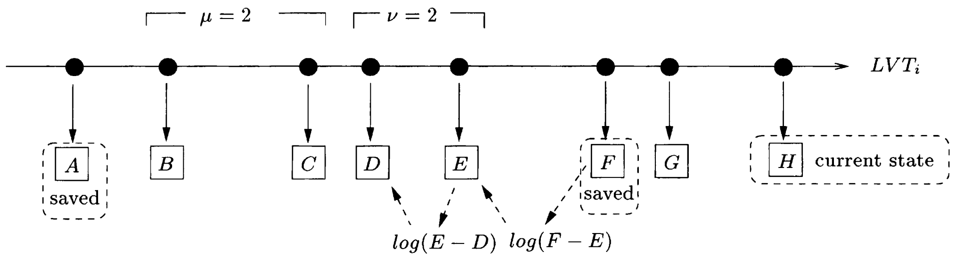

Fig. 5. Mixed state saving.

than that of multiplexed state saving, once fixed the same value for χ in both solutions.

We associate with mixed state saving a recovery procedure which is actually the first solution embedding both coasting forward and backward recovery modes (i.e., they never appear as combined together in previous recovery procedures). The procedure consists of two mutually exclusive parts. One, namely *forward recovery*, represents the case of state recovery through coasting forward, the other, namely *backward recovery*, represents the case of state recovery through the application of logs of before-images. The two parts are discussed below. Before entering the description, let us introduce a simple notation: $\mathrm{succ}(X)$ (resp. $\mathrm{pred}(X)$) denotes the state of LP_i immediately following (resp. preceding) the state X.

4.2.1. Forward recovery

This mode is adopted whenever LP_i must rollback to a state X such that $\log(\mathrm{succ}(X) - X)$ does not exist. It works as follows. If X is saved then the recovery procedure entails reloading X into the current state buffer, else it entails reloading the latest saved state preceding X and replaying intermediate events in a coasting forward. As an example, let us consider the portion of the execution of LP_i shown in Fig. 5. If LP_i must roll back to a saved state, namely A or F, then the recovery procedure simply entails the reloading of that state into the current state buffer. Instead, if LP_i must rollback to any of the states in the set $\{B, C\}$, then the recovery procedure entails reloading A and, in addition, replaying intermediate events to update state variables.

The worst case recovery latency occurs when LP_i must roll back to a non-recorded state X such that: (i) $\log(\mathrm{succ}(X) - X)$ does not exist, and (ii) either $\log(\mathrm{succ}(\mathrm{succ}(X)) - \mathrm{succ}(X))$ exists or $\mathrm{succ}(X)$ is recorded.

In this case (e.g., state C in Fig. 5), all the $\mu = \chi - 1 - v$ events for which before-images were not saved must be replayed, so the worst case recovery latency is proportional to the quantity $\chi - 1 - v$.

4.2.2. Backward recovery

This mode is adopted whenever LP_i must roll back to a state X such that $\log(\mathrm{succ}(X) - X)$ exists. In this case the recovery procedure entails reloading the oldest saved state following X, if any, and backward applying logs of before-images to the state variables. If no state following X is saved, then logs of before-images are applied to the current state of LP_i. For the example shown in Fig. 5, if LP_i must rollback to E then F is reloaded and $\log(F - E)$ is applied.

The worst case recovery latency occurs when LP_i must roll back to a non-recorded state X such that $\log(X - \mathrm{pred}(X))$ does not exist and at least one state following X has been saved (e.g., state D in Fig. 5). In this case, all the v logs of before-images within successive state saving operations must be applied, so the worst case recovery latency is proportional to the quantity v.

From previous discussion, it comes out that the worst case recovery latency of our scheme is proportional to the quantity $\max(\chi - 1 - v, v)$. Therefore, compared to the schemes based on sparse and multiplexed state saving, where the worst case recovery latency is proportional to $\chi - 1$, our scheme shows the potential for faster worst case recovery when considering the same value for χ.

Compared to the scheme based on incremental state saving, our scheme allows the recovery latency to be independent of the rollback extent. As already pointed out in Section 4.1, this independence is a highly desirable feature.

An important feature distinguishing our scheme from previous ones is that the checkpointing protocol has two tunable parameters (i.e., χ and v) instead of either a single one (as in sparse and multiplexed state saving) or zero (as in copy and incremental state saving). So our scheme is more flexible from the performance optimization point of view.

Finally we remark that, by selecting particular values for the parameters χ and v, many of the other checkpointing–recovery schemes can be obtained as instances of our one. In particular:

- when $\chi = 1$ (thus v is necessarily equal to zero) we get the scheme based on copy state saving;
- when $\chi > 1$ and $v = 0$ we get the scheme based on sparse state saving when considering that checkpoints are taken based on the number of executed events; [5]
- when $v = \chi - 1$ and χ is unbounded we get the scheme based on incremental state saving;
- when $v = \chi - 1$ and χ is bounded we get the scheme based on multiplexed state saving.

5. Analysis of the checkpointing overhead and recovery latency

In this section, we present an evaluation of the average checkpointing overhead and the average recovery latency per event of our scheme vs the checkpointing protocol parameters χ and v. Then two regulating algorithms to tune adaptively these parameters are presented. Both the analysis and the regulating algorithms are derived for a generic logical process LP_i.

5.1. The analysis

Denoting with δ_{ss} the average time to save the state of LP_i, and with δ_{bi} the average time for saving the before-images of the state variables during the execution of an event, we get for the average checkpointing overhead ACO associated with each event the following simple expression:

$$\mathrm{ACO} = \frac{1}{\chi}(\delta_{\mathrm{ss}} + v\delta_{\mathrm{bi}}). \tag{1}$$

[5] There exist other solutions for sparse state saving in which a checkpoint is not taken each χ event executions but based on other criteria, namely elapsed time [2] and differences between timestamps of consecutive events [19–21]. The out-coming checkpointing–recovery schemes cannot be considered as instances of our one.

Eq. (1) simply states that for a period of χ successive states, the checkpointing overhead is: 1 saving of the state and v logs of the before-images.

In order to analyze the average recovery latency per event, let us denote with δ_{rl} the average time to reload a saved state into the current state location, with δ_{event} the average event replaying time and with δ_{back} the average time to apply a log of before-images. Our checkpointing protocol selects the states to be recorded and the before-images to be logged based only on the parameters χ and v. So there is no correlation between where in the simulation time state information are recorded and where in the simulation time rollbacks occur. Several studies [8,23] have shown that for any protocol which does not generate such a correlation, all the states between two successively recorded ones have the same probability to be restored due to rollback. Therefore, if a rollback occurs then:

- the probability to execute a forward recovery is $(\chi - v)/\chi$ and the average length of the associated coasting forward is $(\chi - v - 1)/2$. Thereby, the average cost of forward recovery, namely AFR, can be expressed as:

$$\mathrm{AFR} = \frac{\chi - v}{\chi}\left(\delta_{\mathrm{rl}} + \frac{\chi - v - 1}{2}\delta_{\mathrm{event}}\right), \tag{2}$$

- the probability to execute a backward recovery is v/χ and its average length is $v/2$. Thereby, the average cost of backward recovery, namely ABR, can be expressed as:

$$\mathrm{ABR} = \frac{v}{\chi}\left(\delta_{\mathrm{rl}} + \frac{v}{2}\delta_{\mathrm{back}}\right). \tag{3}$$

Denoting with F_{r} the average rollback frequency experienced by LP_i, we get for the average recovery latency ARL per event the following expression:

$$\mathrm{ARL} = F_{\mathrm{r}}[\mathrm{AFR} + \mathrm{ABR}]. \tag{4}$$

Plugging (2) and (3) in (4) we get for ARL the following expression:

$$\mathrm{ARL} = F_{\mathrm{r}}\left[\frac{\chi - v}{\chi}\left(\delta_{\mathrm{rl}} + \frac{\chi - v - 1}{2}\delta_{\mathrm{event}}\right) + \frac{v}{\chi}\left(\delta_{\mathrm{rl}} + \frac{v}{2}\delta_{\mathrm{back}}\right)\right]. \tag{5}$$

The average checkpointing overhead and recovery latency per event, namely ACR, can be expressed as the sum of ACO and ARL. Therefore, combining (1) and (5) we get for ACR the following expression, showing its dependence on the checkpointing protocol parameters χ and v and on the average rollback frequency F_{r}:

$$\mathrm{ACR} = \frac{1}{\chi}(\delta_{\mathrm{ss}} + v\delta_{\mathrm{bi}}) + F_{\mathrm{r}}\left[\frac{\chi - v}{\chi}\left(\delta_{\mathrm{rl}} + \frac{\chi - v - 1}{2}\delta_{\mathrm{event}}\right) + \frac{v}{\chi}\left(\delta_{\mathrm{rl}} + \frac{v}{2}\delta_{\mathrm{back}}\right)\right]. \tag{6}$$

Based on expression (6), we present in the following section two algorithms to dynamically regulate the checkpointing protocol parameters χ and v. These algorithms are based on the gathering of statistics which are used to estimate the average rollback frequency F_r. [6]

5.2. Regulating algorithms

The easiest way to use our scheme, in terms of setting of the checkpointing protocol parameters, consists of assigning fixed values to χ and v at the beginning of the execution and avoiding any further adjustment. Although this approach does not introduce any computational effort, the unpredictable value of the average rollback frequency F_r usually prevents from choosing values for χ and v that keep low the checkpointing–recovery overhead per event. Furthermore, as already outlined, F_r may change in the lifetime of the simulation execution, possibly making unsuitable the initial choice, even in the case good initial values were selected.

We present in this section two algorithms, relying on the analysis performed in Section 5.1, which tune the values of χ and v adaptively in order to keep low the checkpointing–recovery overhead even in the case F_r changes in the lifetime of the execution. As already pointed out, the algorithms are based on the estimate of the rollback frequency F_r, which is usually performed in the following standard way [19,23]. Each process LP_i suspends periodically its execution and estimates the rollback frequency as the ratio between the number of rollback occurrences and the number of event executions. The estimate makes use of statistics related to a temporal window whose length is selected by the application programmer, possibly in a dynamic fashion. This will allow the estimate to be performed using statistical data related only to a recent portion of the simulation. The reader can refer to [23] for more details on this issue.

The algorithms we present are based on the following skeleton:

(A) The estimated value of F_r is plugged in expression (6) and the values $\widehat{\chi}$ and $\widehat{v}$ minimizing ACR over a *feasible domain* for χ and v are computed.
(B) The current values of χ and v are updated based on the computed values $\widehat{\chi}$ and $\widehat{v}$ and a given *updating policy*.

The feasible domain of values for χ and v is determined in a static way as follows. The application programmer selects a maximum value $\chi_{\max}$ [7] determining for χ the feasible range $[1, \chi_{\max}]$. For each value of χ in the range $[1, \chi_{\max}]$, the range of v is straightforwardly defined as the interval $[1,\chi)$ (i.e., $v < \chi$ with v non-negative). Although standard approaches for solving the linear integer programming problem in point (A) encompass branch and bound and others, the limited size of the feasible domain (deriving from those classical values for χ) allows the minimization to be solved as a problem of search of the minimum of a function overall the domain. Results we report in Section 7.2 demonstrate the viability of this solution, pointing out that the overhead of this approach results negligible.

[6] Other quantities involved in Expression (6) can be usually known by analyzing the application code. Therefore they usually do not need to be estimated.

[7] Standard values are between 10 and 30, see [8,20,23].

The following updating policies determine the two distinct regulating algorithms we propose.

5.2.1. Raw policy

The pair of values $\langle \widehat{\chi}, \widehat{v} \rangle$ resulting from the minimization of ACR in step A are rawly assigned to the pair of protocol parameters $\langle \chi, v \rangle$. In other words, the values leading to the absolute minimum of the function ACR are selected for $\langle \chi, v \rangle$. This approach relies on the presumption that the estimated value of F_r used in the minimization is a good indicator of the real rollback frequency of the LP in the immediate future. Empirical evidence [21,23] shows that this is likely to occur in practice whenever the simulation application is not perturbed by external workload from other users of the computing system. This is because (i) simulation intrinsics (e.g., phase behavior of the LPs) typically produces notable changes of the rollback behavior only on long-term basis and, in addition, (ii) changes in the checkpointing protocol parameters typically do not perturb the rollback behavior of the LP in practice [21,23].

5.2.2. Smooth policy

Differently from the raw policy, updates of $\langle \chi, v \rangle$ are limited into the range $\langle \chi \pm 1, v \pm 1 \rangle$. In other words, each protocol parameter is updated through a unit increment (resp. decrement) if it is smaller (resp. larger) than the corresponding value computed in the minimization in step A. As an example, if the current values of χ and v are 3 and 1, respectively, and the values minimizing ACR computed in step A are $\widehat{\chi} = 5$ and $\widehat{v} = 3$, then the new value assigned to χ is 4 and the new value assigned to v is 2 (i.e., χ and v are moved of 1 unit towards the values $\widehat{\chi}$ and $\widehat{v}$).

This solution is likely to be suited for all the cases in which there could be strong interference from external workloads which may produce shocks in the rollback behavior of the LPs [7]. In these cases, the minimization of the value of ACR is done using an estimated value for F_r that could be loosely related to the immediate future rollback rate of the LP. Therefore, rawly moving $\langle \chi, v \rangle$ to the values $\langle \widehat{\chi}, \widehat{v} \rangle$ is likely to not yield good performance.

On the other hand, the drawback here leans on the strong dependence of the performance on the initial values assigned to the pair $\langle \chi, v \rangle$. In case of bad initial values, even for a stable behavior of the rollback rate, it may happen that the smooth changes of the protocol parameters lead them to asymptotically tend towards adequate values, without ever reaching them. In order to overcome this problem, the first update of the protocol parameters assigns the values $\langle \widehat{\chi}, \widehat{v} \rangle$ to $\langle \chi, v \rangle$. Doing so, unless shocks occur in the early phase of the simulation execution, problems due to possible bad choice of the initial values are removed.

6. Implementation issues

In this section, we discuss issues related to the implementation of our checkpointing–recovery scheme. Our state recovery procedure does not require complex operations; it only needs the managing of buffers containing recoded states or logs of before-images and, possibly, the managing of the event-queue in case of forward recovery. Instead, the problematic part of the scheme is the checkpointing protocol, as it requires special handling for any event for which before-images of the state variables have to be saved during its execution. For this reason, our discussion focuses on requirements to support mixed state saving.

Basically, the implementation we propose relies on the presence of two distinct event execution procedures, namely *event_ex*() and *event_ex_log*(). The procedure *event_ex*() is called whenever an event e has to be executed and no before-image has to be logged during the execution. *event_ex_log*() is instead called whenever before-images have to be logged during the execution. All the statements in *event_ex*() are replicated in *event_ex_log*(), but *event_ex_log*() is augmented with saving calls of the before-images.

In Fig. 6, the pseudo-code for LP$_i$ is reported. We focus from row 9 on, as the preceding rows are self-explaining. The procedure *protocol_decision*() implements part of the checkpointing protocol. It checks the need for either saving the state of LP$_i$ prior the execution of the next event e, or logging before images during the execution of e. The result of the decision is recorded into the variable *action_type*. The procedure *save_state*(), called in case of checkpoint before the event execution, performs the state saving operation of the whole state of LP$_i$. The procedure *recompute_parameters*() tests the need for recomputing the checkpointing protocol

```
program LP_i()
1     initialize();
2     while GVT ≤ end_time do
3        <collect all incoming messages/antimessages and update the event-queue>
4        if rollback
5              <recovery to the earliest state prior the rollback point>;
6              <send-out antimessages>;
7        endif;
8        <extract the next event e to be executed>;
9        action_type = protocol_decision()
10       case action_type of
11          no_action : event_ex();
12          checkpoint : save_state(); event_ex();
13          before_image_log : event_ex_log();
14       endcase;
15       <send-out produced events>;
16       <advance GVT and recover memory>;
17       recompute_parameters();
18    endwhile
```

Fig. 6. The pseudo-code for LP$_i$.

parameters and, if needed, recalculates them according to the selected policy. Typically, the recalculation is executed every few hundreds through few thousands events (see [8,20,23]), therefore the task of *recompute_parameters*() entails to check whether such amount of events have been executed by the LP from the last recalculation.

The core problem for the generation of the procedure *event_ex_log*() is related to the insertion of saving calls of before-images of the state variables. We briefly discuss advantages and disadvantages of the three classical methods to solve this problem, namely the manual insertion, the use of modified compilers and the use of augmented data types.

In the manual insertion, saving calls are placed into the code by the application programmer who is responsible for tracking all the updates on state variables. Although such solution is quite simple, it shows the drawback of absence of transparency. Furthermore, it is an error prone approach anytime the application programmer has no special understanding of state saving methods. In such a case, even if a single state variable is not recorded prior updating it, then the recovery procedure may produce incorrect states.

A modified compiler has to recognize instructions which modify state variables and insert instructions to save their values before they are updated. This solution improves the transparency to the application programmer. However, it shows the drawback that sometimes libraries which are not compiled with the modified compiler cannot be used. In addition, the modified compiler could require huge programming effort for being implemented.

The third solution, namely the use of augmented data types, is such that saving calls of before-images are encapsulated in the type definition. [8] Again, this approach requires that the programmer, who is responsible for the data type definition/construction, has deep knowledge of state saving mechanisms.

Previous discussion points out that, compared to copy or sparse state saving (which do not need insertion of saving calls of before-images), mixed state saving shows lower transparency to the application programmer, just like incremental and multiplexed state saving. On the other hand, we believe that some programming effort should be devoted to the production of optimized simulation software allowing faster execution.

7. Performance evaluation

In this section, we report performance data for a comparison between our scheme and previous solutions. We first introduce the hardware/software platform and the benchmark we have used. Then we present the performance indices and the schemes we have selected for the comparison. Finally, the performance data are reported and discussed.

[8] This is particularly simple to implement when using an object oriented language, that allows the construction of objects with hidden state saving behavior and with external behavior (and appearance) of a primitive data type of the language.

7.1. The hardware/software platform and the benchmark

As hardware architecture we used a cluster of machines (Pentium 233 MHz – 64 Mbytes RAM) connected via Ethernet. Inter-processor communication relies on message passing supported by PVM [30]. A Time Warp kernel runs on each processor. The kernel schedules LPs for running according to the Smallest-Timestamp-First algorithm [14]. GVT is computed periodically, and memory allocated for obsolete information is recovered periodically as well.

As benchmark we consider the PHOLD model [11]. It consists of a fixed number of LPs, each one modeling a service center, and of a constant number of customers circulating among the LPs. Each service center is assumed to have an infinite number of servers, therefore a customer entering the center immediately receives the service. (In other words, the PHOLD model represents a queuing network with an infinite number of severs for each queue.) After it is served, the customer is forwarded to another center. In our tests the destination center is selected according to an uniform distribution, therefore customers are equally likely to be forwarded to any center (i.e., to any LP). The service time, determining the timestamp increment for the forwarded customer, follows an exponential distribution with mean 1 simulated time unit. The customer population has been fixed at 10 customers per service center.

We choose this benchmark for two main reasons: (i) its parameters (e.g., event processing time, size of the LP state, etc.) can be easily modified, (ii) it is the most used benchmark for testing performance of checkpointing–recovery schemes [4,19,21,23,27,32] and scheduling algorithms [22,25] for the Time Warp mechanism, and of synchronization mechanisms derived from Time Warp by imposing limitations on the optimism of event execution [28]. Furthermore, the PHOLD model shows a rollback behavior similar to many other synthetic benchmarks and also to several real world models, therefore a wide class of simulation models should lead to performance data quite similar to those obtained with it.

We have considered two different pairs of values for the number of LPs constituting the simulation model and the number of processors used to simulate the model (i.e., $\langle 32, 8 \rangle$ and $\langle 16, 16 \rangle$); in both cases, each processor runs the same number of LPs. Therefore the pair $\langle 32, 8 \rangle$ gives rise to a simulation with limited degree of parallelism (each processor runs four LPs), whereas the pair $\langle 16, 16 \rangle$ gives rise to a simulation with high degree of parallelism (each processor runs only one LP).

For each pair, two different values of the ratio α between the average event execution time δ_{event} and the average time to save the state δ_{ss} have been considered, i.e., $\alpha = 1/2$ and $\alpha = 2$. The case $\alpha = 1/2$ is representative of simulations with large state granularity. It has been obtained by fixing the state saving time δ_{ss} at 300 μs, corresponding to the average time to save a fictitious state of 8 Kbytes, and (i) either fixing δ_{event} at 150 μs (such a value is obtained by introducing a fixed delay loop into the event routine), (ii) or by extracting δ_{event} from an exponential distribution with mean 150 μs (in this case any event has its proper delay loop length determined according to such stochastic distribution). The case $\alpha = 2$ is representative of simulations with medium/small state granularity; it has been obtained by fixing δ_{ss} at

Table 1
The eight basic configurations

$\langle \text{LPs}, proc\rangle$ =	$\langle 32, 8\rangle$		$\langle 16, 16\rangle$	
β =	0.3	0.7	0.3	0.7
$\alpha = 1/2$	C1	C2	C5	C6
$\alpha = 2$	C3	C4	C7	C8

150 μs, corresponding to the average time to save a fictitious state of 4 Kbytes, and (i) either fixing δ_{event} at 300 μs, (ii) or extracting δ_{event} from an exponential distribution with mean 300 μs. In other words, for each value of α we report results for both the case of deterministic and stochastic event granularity. For each out-coming configuration we have considered two different percentages of the state updated by any event execution, namely $\beta = 0.3$ and $\beta = 0.7$. These eight basic configurations of the benchmark are summarized in Table 1.

Furthermore, to stress the behavior of our scheme when minimal and maximal fractions of the state are updated by event execution, we report results for four additional configurations. The first two are related to the case $\langle 32, 8\rangle$ and are characterized by the value $\beta = 0.1$ (minimal fractions of the state are updated) and by the value $\beta = 0.9$ (maximal fractions of the state are updated), respectively. The latter two are related to the case $\langle 16, 16\rangle$ and, as for previous case, are characterized by the values $\beta = 0.1$ and $\beta = 0.9$, respectively.

For these additional configurations we have selected the value $\alpha = 4$ in order to include in our study an additional point for this parameter. Such a value has been obtained by fixing δ_{ss} at 100 μs and extracting δ_{event} from an exponential distribution with mean 400 μs. With these additional configurations we have actually covered a reasonably wide range of values for the parameters β and α. [9] The four additional configurations of the benchmark are summarized in Table 2.

For all the experiments, one additional process, not belonging to the set of LPs constituting the simulation model, is always running on the cluster, and randomly migrates among the machines each 2.5 s. The performance of the machine hosting this process is temporarily halved; in particular, the additional process consumes half of the CPU time of the machine it runs on. Therefore, the LPs on that machine take, on average, a doubled amount of real time to execute any of their operations. The additional process is introduced in order to get non-stationary rollback behavior of the simulation.

[9] We would like to stress that values of α less than 1/2 are very unusual in Time Warp simulations since for simulations where state saving time is strongly predominant over the event execution time it is preferable to adopt the sequential simulation approach or, in case of non-minimal lookahead in the simulation model, the conservative parallel approach. In both these approaches state saving is not required. Furthermore, for the case of very large values of α (e.g., in the order of 10 or more), the checkpointing–recovery scheme based on copy state saving results well suited (since state saving before any new event produces negligible overhead), thus removing the need for other optimized checkpointing–recovery schemes.

Table 2
The four additional configurations

$\langle \text{LPs}, proc \rangle =$	$\langle 32, 8 \rangle$		$\langle 16, 16 \rangle$	
$\beta =$	0.1	0.9	0.1	0.9
$\alpha = 4$	C9	C10	C11	C12

7.2. Performance indices and schemes selected for the comparison

We report measures related to the following performance indices:

- the *event rate*, that is the average number of committed events per second (this parameter indicates how fast is the simulation execution with a given checkpointing scheme);
- the *average checkpointing overhead*, that is the average time spent for state saving operations per event;
- the *average recovery latency*, that is the average time to reconstruct the state to which the LP has to roll back in case of timestamp order violation;
- the *memory usage*, that is the average amount of bytes allocated for state information per second.

We compare our checkpointing–recovery scheme (hereafter CQ) to the following ones:

- Ronngren and Ayani's scheme (RA) [23], and Fleischmann and Wilsey's scheme (FW) [8]. Both of them rely on sparse state saving; more precisely, the state is saved based on the number of simulation events (i.e., each χ event executions). RA dynamically recalculates the value of χ based on an analytical model of the LP execution time; the model identifies the value of the time-optimal checkpoint interval for the estimated value of the rollback frequency F_r. FW selects an initial value and an adaptation direction for the checkpoint interval χ (the adaptation step is 1); then, the adaptation direction is inverted each time the monitored checkpointing–recovery overhead experienced by the LP shows a significant increase (so this solution does not rely on the estimation of F_r).
  Note that recently it has been shown in [19–21] that, for some particular simulation problems with regular patterns for the difference between timestamps of successive events, it is possible to select checkpoint positions leading to lower recovery time compared to RA and FW. However, the sparse state saving approach based on the number of simulation events (i.e., the one underlying both RA and FW) remains, in general, the best solution;
- the scheme based on incremental state saving (INCR);
- the scheme based on multiplexed scheme (MUL); for this scheme we report data obtained with the value of the checkpoint interval χ yielding the highest observed event rate. [10]

[10] $\chi = \infty$ means that the highest event rate has been observed for MUL behaving as INCR.

For the schemes RA, FW and CQ the checkpointing protocol parameters are recalculated for any LP each 100 events (F_r is estimated in RA and CQ using statistics related to the last 300 events). [11] Furthermore, for the scheme CQ, the setting $\chi_{max} = 10$ is adopted in order to keep reasonably small the minimization domain of expression (6). For such a value we have measured an overhead due to the minimization procedure (implemented as a problem of search of the absolute minimum overall the feasible domain for the parameters χ and v) of about 25 μs on the used machines. This points out that the additional overhead per event due to the minimization procedure is actually negligible in practice (i.e., about 0.25 μs per event since the recalculation is executed each 100 events). Finally, for CQ performance data for both the regulating algorithms presented in Section 5.2 are reported.

7.3. Performance data

Tables 3–14 show the values of the performance indices measured for all the configurations of Table 1 and Table 2 (each value reported is the average over 10 runs all done with different random seeds). The *average checkpointing overhead* and the *average recovery latency* are expressed in microseconds, the *event rate* in events/s, and the *memory usage* in Kbytes/s.

For what concerns the final performance perceived, namely the event rate, the obtained results point out a major tendency regardless of the type of event granularity (deterministic vs stochastic), the ratio between the average state saving time and the average event execution time, and the fraction (minimal, maximal or intermediate) of the state updated by the event execution.

This tendency shows that, for all the benchmark configurations used, no other scheme exhibits notably better performance than that of CQ and, at the same time, depending on the benchmark configuration CQ definitely outperforms one or more of the other schemes. In particular, for configurations C1–C4, CQ coupled with a raw policy to update the checkpointing protocol parameters provides performance which is similar to (or slightly better than) that provided by RA, and performs definitely better than all the other schemes. For configurations C5 and C7, CQ outperforms RA and FW (with a performance gain up to 8%), and provides performance slightly better than that of MUL and INCR. For configurations C6 and C8, CQ definitely outperforms INCR and RA, and exhibits slightly better performance than that of FW and MUL.

Finally, for the case of the additional configurations (C8–C12), we have that when the degree of parallelism is limited (configurations C8 and C9), the only schemes that show performance similar to that of CQ are RA and MUL; all the other schemes perform definitely worse. This happens regardless of the fraction of

[11] Few hundreds of events have been indicated in [23] to be a window length which usually guarantees collected statistical data to be meaningful.

Table 3
Performance indices for configuration C1

Configuration C1	Deterministic event granularity				Stochastic event granularity			
Scheme	Event rate	Recovery latency	Check-pointing overhead	Memory usage	Event rate	Recovery latency	Check-pointing overhead	Memory usage
CQ (raw)	11,270	435.93	48.67	19,825	11,137	422.49	48.60	19,787
CQ (smooth)	10,975	450.84	48.90	19,724	10,851	426.16	49.63	20,035
RA	11,092	556.60	40.05	16,139	10,968	576.95	36.89	14,848
FW	10,678	625.46	51.59	20,266	10,357	757.53	47.34	18,365
INCR	9516	278.85	92.25	34,589	9443	274.46	92.20	34,561
MUL ($\chi = \infty$–∞)	9428	281.37	91.16	34,172	9387	277.54	90.98	34,170

Table 4
Performance indices for configuration C2

Configuration C2	Deterministic event granularity				Stochastic event granularity			
Scheme	Event rate	Recovery latency	Check-pointing overhead	Memory usage	Event rate	Recovery latency	Check-pointing overhead	Memory usage
CQ (raw)	11,190	446.42	47.42	19,308	11,083	451.67	45.88	18,637
CQ (smooth)	10,879	461.55	47.97	19,287	10,757	459.91	46.59	18,733
RA	11,092	556.60	40.05	16,139	10,968	576.95	36.89	14,848
FW	10,678	625.46	51.59	20,266	10,357	757.53	47.34	18,365
INCR	7404	627.31	215.24	61,716	7295	621.07	217.28	61,559
MUL ($\chi = \infty$–∞)	7253	640.25	211.01	60,334	7188	638.13	212.67	60,604

Table 5
Performance indices for configuration C3

Configuration C3	Deterministic event granularity				Stochastic event granularity			
Scheme	Event rate	Recovery latency	Check-pointing overhead	Memory usage	Event rate	Recovery latency	Check-pointing overhead	Memory usage
CQ (raw)	8037	211.87	46.53	14,626	7871	207.49	48.40	14,445
CQ (smooth)	7869	241.77	46.45	14,488	7662	219.27	49.13	14,556
RA	7997	612.11	32.51	9836	7837	708.78	28.95	9250
FW	7512	692.37	47.98	13,900	7275	729.16	43.67	12,408
INCR	7611	148.02	46.12	14,521	7529	149.87	48.37	14,449
MUL ($\chi = \infty$–∞)	7407	151.11	45.62	14,036	7381	153.11	47.04	14,203

the state updated (minimal or maximal). Instead, when the degree of parallelism is increased (configurations C10 and C12), CQ performs definitely better than RA and FW (up to 10%) and shows performance quite similar to that of MUL and INCR.

Table 6
Performance indices for configuration C4

Configuration C4	Deterministic event granularity				Stochastic event granularity			
Scheme	Event rate	Recovery latency	Check-pointing overhead	Memory usage	Event rate	Recovery latency	Check-pointing overhead	Memory usage
CQ (raw)	8024	383.88	45.70	14,049	7919	382.96	47.68	13,905
CQ (smooth)	7786	422.18	45.35	13,779	7661	391.51	48.25	13,950
RA	7997	612.11	32.51	9836	7837	708.78	28.95	9250
FW	7512	692.37	47.98	13,900	7275	729.16	43.67	12,408
INCR	6845	328.34	107.63	29,968	6712	334.09	112.87	29,663
MUL ($\chi = 40$–43)	6900	313.59	106.31	29,761	6846	316.24	110.72	29,804

Table 7
Performance indices for configuration C5

Configuration C5	Deterministic event granularity				Stochastic event granularity			
Scheme	Event rate	Recovery latency	Check-pointing overhead	Memory usage	Event rate	Recovery latency	Check-pointing overhead	Memory usage
CQ (raw)	13,471	318.65	61.86	44,963	12,963	315.08	62.01	44,907
CQ (smooth)	13,335	316.72	61.76	45,011	12,834	306.07	62.67	45,312
RA	12,507	490.78	43.54	30,800	11,090	490.91	43.59	30,725
FW	12,946	436.21	58.25	41,118	11,113	428.24	59.95	41,738
INCR	12,026	310.68	92.25	62,971	11,785	305.20	92.24	62,910
MUL ($\chi = 6$–5)	13,007	180.66	114.71	79,336	12,121	164.46	132.36	82,635

Table 8
Performance indices for Configuration C6

Configuration C6	Deterministic event granularity				Stochastic event granularity			
Scheme	Event rate	Recovery latency	Check-pointing overhead	Memory usage	Event rate	Recovery latency	Check-pointing overhead	Memory usage
CQ (raw)	13,010	361.89	57.94	41,663	12,676	364.46	57.26	41,107
CQ (smooth)	13,426	365.51	57.73	41,884	12,662	363.62	57.24	41,061
RA	12,507	490.78	43.54	30,800	11,090	490.91	43.59	30,725
FW	12,946	436.21	58.25	41,118	11,113	428.24	59.95	41,738
INCR	8841	712.40	215.25	1,10,710	8765	713.05	215.23	1,11,090
MUL ($\chi = 2$–2)	11,228	122.72	247.86	1,42,311	10,829	245.87	248.14	1,42,880

With respect to memory usage, the schemes that tend to exhibit the lowest values are RA and, for some particular configurations with minimal fractions of the state updated by event execution, MUL and INCR. However, although non-minimal for most of the configurations, the memory usage of CQ tends to remain in the 150%, or

Table 9
Performance indices for configuration C7

Configuration C7	Deterministic event granularity				Stochastic event granularity			
Scheme	Event rate	Recovery latency	Check-pointing overhead	Memory usage	Event rate	Recovery latency	Check-pointing overhead	Memory usage
CQ (raw)	11,587	170.55	50.04	30,054	11,003	167.77	52.61	29,892
CQ (smooth)	11,613	167.51	50.17	30,205	11,042	167.24	52.72	30,133
RA	10,718	529.36	35.12	19,817	9755	561.31	34.29	18,740
FW	10,780	378.90	59.21	33,593	9633	441.66	53.36	32,303
INCR	11,540	152.61	46.17	27,987	10,965	159.48	48.37	27,769
MUL ($\chi = \infty$–∞)	11,594	152.37	46.63	28,634	10,982	155.48	47.22	28,703

Table 10
Performance indices for configuration C8

Configuration C8	Deterministic event granularity				Stochastic event granularity			
Scheme	Event rate	Recovery latency	Check-pointing overhead	Memory usage	Event rate	Recovery latency	Check-pointing overhead	Memory usage
CQ (raw)	11,286	290.16	57.31	33,240	10,559	290.52	60.22	32,993
CQ (smooth)	11,331	289.39	57.10	33,171	10,653	290.53	59.98	32,935
RA	10,718	529.36	35.12	19,817	9755	561.31	34.29	18,740
FW	10,780	378.90	59.21	33,593	9633	441.66	53.36	32,303
INCR	9644	349.00	107.62	57,013	9595	359.65	112.87	56,728
MUL ($\chi = 2$–2)	10,815	80.73	123.79	68,606	10,659	83.27	122.31	68,481

Table 11
Performance Indices for Configuration C9

Configuration C9	Stochastic event granularity			
Scheme	Event rate	Recovery latency	Checkpointing overhead	Memory usage
CQ (raw)	6845	92.58	15.44	6686
CQ (smooth)	6783	102.36	15.65	6748
RA	6714	675.34	31.09	11,064
FW	5996	491.11	76.62	28,622
INCR	6696	41.14	10.25	4495
MUL ($\chi = \infty$)	6683	42.76	10.34	4544

less, of the minimal memory usage observed. This should allow CQ to not adversely affect performance when memory is a critical resource.

As last point to note, we have that MUL and INCR show good performance, as compared to CQ, for the cases in which the fraction of the state updated is minimal, see C7, C9 and C11. The reason for this behavior is in that when the fraction of the state updated is minimal these schemes provide very low checkpointing overhead per

Table 12
Performance indices for configuration C10

configuration C10	stochastic event granularity			
Scheme	Event rate	Recovery latency	Checkpointing overhead	Memory usage
CQ (raw)	6730	285.67	44.15	17,897
CQ (smooth)	6595	293.81	44.16	17,805
RA	6714	675.34	31.09	11,064
FW	5996	491.11	76.62	28,622
INCR	5998	284.78	92.24	34,747
MUL ($\chi = 3$)	6267	283.87	95.84	35,955

Table 13
Performance indices for configuration C11

Configuration C11	Stochastic event granularity			
Scheme	Event rate	Recovery latency	Checkpointing overhead	Memory usage
CQ (raw)	10,297	84.92	16.27	13,742
CQ (smooth)	10,224	84.46	16.29	13,738
RA	8934	601.18	26.09	19,567
FW	9422	22.23	100.74	77,159
INCR	10,274	42.47	10.25	8796
MUL ($\chi = \infty$)	10,251	41.45	10.88	8843

Table 14
Performance indices for configuration C12

Configuration C12	Stochastic event granularity			
Scheme	Event rate	Recovery latency	Checkpointing overhead	Memory usage
CQ (raw)	9677	248.79	47.89	37,467
CQ (smooth)	9647	248.12	47.97	37,538
RA	8934	601.18	26.09	19,567
FW	9222	482.23	43.74	33,159
INCR	9030	300.75	92.24	67,494
MUL ($\chi = 2$)	9145	276.45	94.34	67,986

event. On the other hand, the checkpointing overhead of CQ for these configurations results slightly larger due to the fact that we have imposed the limitation $\chi_{\max} = 10$ in order to keep low the amount of time required for the minimization of the ACR function.

Overall, CQ shows the potential to provide good performance for a wider set of simulation settings as compared to all the other schemes tested, thus representing a general solution for tackling the checkpointing–recovery problem in Time Warp simulations.

8. Summary

In this paper a checkpointing–recovery scheme for Time Warp parallel simulations has been introduced. The scheme consists of a checkpointing protocol, resulting as a modification of an existing one, and an associated recovery procedure embedding both classical forward and backward recovery modes. The checkpointing protocol is the first one having two tunable parameters, instead of a single one or, at worst, none. This feature infers to it more flexibility than previous ones from the performance optimization point of view. Furthermore, it allows the scheme to play the role of a generalization of several previous schemes. Two regulating algorithms to adaptively tune the checkpointing protocol parameters are also presented. They are based on an analysis we perform of the checkpointing overhead and the recovery latency of our scheme.

In order to demonstrate the effectiveness of our scheme, several performance indices have been measured for a classical benchmark in several different configurations. The results show that our solution allows faster execution and keeps quite low the memory usage for any of the configurations, thus exhibiting the potential for resulting well suited to a wide class of simulation applications.

References

[1] H. Bauer, C. Sporrer, Reducing rollback overhead in Time Warp based distributed simulation with optimized incremental state saving, in: Proceedings of the 26th Annual Simulation Symposium, March 1992, pp. 12–20.

[2] S. Bellenot, State skipping performance with the Time Warp operating system, in: Proceedings of the Sixth Workshop on Parallel and Distributed Simulation (PADS'92), January 1992, pp. 33–42.

[3] J. Briner Jr., Fast parallel simulation of digital systems, in: Proceedings of the Multiconference Advances in Parallel and Distributed Simulation, vol. 23 (1), 1991, pp. 71–77.

[4] D. Bruce, The treatment of state in optimistic systems, in: Proceedings of the Nineth Workshop on Parallel and Distributed Simulation (PADS'95), June 1995, pp. 40–49.

[5] C.D. Carothers, R.M. Fujimoto, Y.B. Lin, P. England, Distributed Simulation of Large Scale PCS Networks, in: Proceedings of the Second International Conference on Modeling and Simulation of Computer and Telecommunication Systems (MASCOTS'94), January 1994, pp. 2–11.

[6] K. Chandy, J. Misra, A case study in the design and verification of distributed programs, IEEE Trans. Software Eng. 5 (5) (1979) 440–452.

[7] A. Ferscha, J. Johonson, Shock resistant Time Warp, in: Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99), May 1999, pp. 92–100.

[8] J. Fleischmann, P.A. Wilsey, Comparative analysis of periodic state saving techniques in Time Warp simulators, in: Proceedings of the Nineth Workshop on Parallel and Distributed Simulation (PADS'95), June 1995, pp. 50–58.

[9] S. Franks, F. Gomes, B. Unger, J. Cleary, State saving for interactive optimistic simulation, in: Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS'97), June 1997, pp. 72–79.
[10] R.M. Fujimoto, Time warp on a shared memory multiprocessor, Trans. Soc. Comput. Simul. 6 (3) (1989) 211–239.
[11] R.M. Fujimoto, Performance of Time Warp under synthetic workloads, in: Proceedings of the Multiconference Distributed Simulation, Vol. 22 (1), 1990.
[12] R.M. Fujimoto, Parallel discrete event simulation, Commun. ACM 33 (10) (1990) 30–53.
[13] R.M. Fujimoto, G.C. Gopalakrishnan, Design and evaluation of the rollback chip: special purpose hardware for Time Warp, IEEE Trans. Comput. 41 (1) (1992) 68–82.
[14] D.R. Jefferson, Virtual time, ACM Trans. Program. Language Syst. 7 (3) (1985) 404–425.
[15] Y.B. Lin, B.R. Preiss, W.M. Loucks, E.D. Lazowska, Selecting the checkpoint interval in Time Warp simulation, in: Proceedings of the Seventh Workshop on Parallel and Distributed Simulation (PADS'93), May 1993, pp. 3–10.
[16] A.C. Palaniswamy, P.A. Wilsey, An analytical comparison of periodic checkpointing and incremental state saving, in: Proceedings of the Seventh Workshop on Parallel and Distributed Simulation (PADS'93), May 1993, pp. 127–134.
[17] A.C. Palaniswamy, P.A. Wilsey, Adaptive checkpoint intervals in an optimistically synchronized parallel digital system simulator, in: Proceedings of the IFIP TC/WG10.5 International Conference on Very Large Scale Integration, September 1993, pp. 353–362.
[18] M. Presley, M. Ebling, F. Wieland, D.R. Jefferson. Benchmarking the Time Warp operating system with a computer network simulation, in: Proceedings of the SCS Multiconference on Distributed Simulation, vol. 21 (2), 1989, 8–13.
[19] F. Quaglia, Event history based sparse state saving in Time Warp, in: Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS'98), May 1998, pp. 72–79.
[20] F. Quaglia, Combining periodic and probabilistic checkpointing in optimistic simulation, in: Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99), May 1999, pp. 109–116.
[21] F. Quaglia, A cost model for selecting checkpoint positions in Time Warp parallel simulation, Technical Report 12-99, DIS, Università di Roma La Sapienza, April 1999 (to appear in IEEE Trans. Parallel Distributed Syst.).
[22] F. Quaglia, V. Cortellessa, Grain sensitive event scheduling in Time Warp parallel discrete event simulation, in: Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS'00), May 2000, pp. 173–180.
[23] R. Ronngren, R. Ayani, Adaptive checkpointing in Time Warp, in: Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS'94), May 1994, pp. 110–117.
[24] R. Ronngren, M. Liljenstam, R. Ayani, J. Montagnat, Transparent incremental state saving in Time Warp parallel discrete event simulation, in: Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96), May 1996, pp. 70–77.
[25] T.K. Som, R.G. Sargent, A probabilistic event scheduling policy for optimistic parallel discrete event simulation, in: Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS'98), May 1998, pp. 56–63.
[26] H.M. Soliman, A.S. Elmaghraby, An analytical model for hybrid checkpointing in Time Warp distributed simulation, IEEE Trans. Parallel Distributed Syst. 9 (10) (1998) 947–951.
[27] S. Skold, R. Ronngren, Event sensitive state saving in Time Warp parallel discrete event simulations, in: Proceedings of 1996 Winter Simulation Conference, December 1996.
[28] S. Srinivasan, P.F. Reynolds Jr., Elastic time, ACM Trans. Modeling Comput. Simul. 8 (2) (1998) 103–139.
[29] J. Steinman, Incremental state saving in SPEEDS using C plus plus, in: Proceedings of 1993 Winter Simulation Conference, December 1993, pp. 687–696.
[30] V.S. Sunderam, A framework for parallel distributed computing, Concurrency: Practice Experience 2 (4) (1990).

[31] B.W. Unger, J.G. Cleary, A. Covington, D. West, External state management system for optimistic parallel simulation, in: Proceedings of 1993 Winter Simulation Conference, December 1993, pp. 750–755.

[32] D. West, K. Panesar, Automatic incremental state saving, in: Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96), May 1996, pp. 78–85.

[33] F. Wieland , L. Hawley, A. Feinberg, M. DiLoreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, D.R. Jefferson, Distributed combat simulation and Time Warp the model and its performance, in: Proceedings of the SCS Multiconference on Distributed Simulation, vol. 21 (2) 1989, 14–20.