A State-Based Scheduling Algorithm for Time Warp Synchronization

FRANCESCO QUAGLIA
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Via Salaria 113, 00198 Roma, Italy
e-mail: quaglia@dis.uniroma1.it

Abstract

This paper presents a state-based scheduling algorithm for the selection of the next logical process (LP) to be run on a processor in a Time Warp synchronized parallel discrete event simulation. In our solution, state information related to the LPs in the immediate predecessor set of a given LP is used to compute its scheduling priority. This distances our algorithm from previous solutions where the scheduling priority is assigned basing exclusively on local state information related to the LPs sharing the processor.

As a system to spread the required state information we use a classical piggybacking technique (i.e. state information is attached to any message carrying a simulation event/antievent). This solution adds negligible overhead but does not prevent state information from becoming stale. To tackle staleness we introduce a notion of information filtering and present an iterative procedure for the selection of an adequate value for the filter length that determines both the amount of (and also what) state information is actually relevant for computing the priority of any LP.

An empirical study of a classical benchmark is reported for a comparison with the Lowest-Timestamp-First algorithm. The obtained data point out the effectiveness of our algorithm in the reduction of the amount of rollback which, in turn, leads to faster execution of the simulation.

1 Introduction

In parallel discrete event simulation, distinct parts of the system to be simulated are modeled by distinct logical processes (LPs) having their own local simulation clocks [4], namely Local Virtual Times (LVTs). An LP executes a sequence of events, and each event execution possibly produces new events to be executed at later simulation time. LPs notify events among each other by exchanging messages carrying the content and the occurrence time (timestamp) of the event. In order to ensure correct simulation results, synchronization mechanisms are used to maintain a non-decreasing timestamp order for the execution of events at each LP.

In Time Warp synchronization [7], LPs are allowed to execute events greedily (i.e. as soon as they are available), by optimistically assuming that no timestamp order violation will occur. If one such violation is detected (i.e. an LP receives a message carrying an event with timestamp lower than that of some already executed event), a rollback procedure recovers the LP state to its value immediately prior to the violation. While rolling back, the LP undoes the effects of the events produced during the rolled back portion of the simulation. This is done by sending a message with an antievent for each event that must be undone. Upon the receipt of an antievent corresponding to an already executed event, the recipient LP rolls back as well. (In the remainder of the paper, we will refer to any message carrying either an event or an antievent as a simulation message).

One of the major shortcomings of Time Warp synchronization is that it may originate poor performance in the case of large amount of rollback. Currently the main strategy for keeping low the amount of rollback consists of controlling (i.e. imposing a limitation on) the optimism of event execution. This solution includes throttling [2, 3, 21], time windows [10, 19, 22, 24], and others [1, 11]. However, some papers [12, 16, 18] have shown that the amount of rollback can be kept low also by working at the level of the scheduling algorithm for the selection of the next event to be executed on a processor. Basically these papers propose scheduling algorithms alternative to the Lowest-Timestamp-First (LTF) algorithm [8], which is widely recognized as a standard solution.

In this paper we design a state-based scheduling algorithm suited for Time Warp synchronization. The state information used by the algorithm is a non-definitive lower bound on timestamps of events that will be produced by an LP in the future of the simulation execution. The term “non-definitive” indicates that the lower bound may be invalidated, but this may happen only in case of rollback of the LP. The values of non-definitive lower bounds of the LPs in the immediate predecessor set of a given LP, together with the timestamp of its next event, are used to establish the priority (high or low) of the LP (\(^1\)). Specifically, high priority is assigned to any LP whose next event has timestamp lower than non-definitive lower bounds of the LPs in its immediate predecessor set. As such an event can be rolled back only conditional at least one of those non-definitive lower bounds is revoked, that is conditional a rollback occurs on one LP in the immediate predecessor set, the scheduling algorithm will possibly reduce the amount of rollback.

The spreading of non-definitive lower bounds, i.e. state information, among the LPs needs a proper system. In our solution, such a spreading is realized through a class-

\(^1\)The immediate predecessor set of an LP consists of all the LPs that can send a simulation message to that LP. As a extreme, this set could contain all the LPs of the simulation.
sical piggybacking technique, that is whenever an LP sends a simulation message \( m \) to any other LP, its current non-definitive lower bound is piggybacked on \( m \). This solution has the advantage of negligible overhead but does not prevent state information from becoming stale. To tackle staleness we propose an iterative procedure for the selection of an adequate amount of the most recently updated state information which must be considered as relevant to establish the priority of a given LP. The procedure is based on the monitoring of the variations of the event rate (i.e. committed events per second), which is representative of the speed of the simulation execution. As an extreme, the procedure may establish that no state information must be taken into account in the scheduling decision. In this case the algorithm behaves exactly like LTF. This points out the generality of our solution that has the capability to produce better or at least the same performance of a standard scheduling algorithm.

The remainder of the paper is organized as follows. In Section 2 a brief overview of existing scheduling algorithms is given. In Section 3 we discuss how to compute non-definitive lower bounds and describe the piggybacking technique to spread them among the LPs. Section 4 presents the scheduling algorithm. Performance data for a comparison between our algorithm and the classical LTF algorithm are reported in Section 5.

2 Related Work

In simulations of large and complex systems it is likely that the number of LPs exceeds the number of processors, consequently each processor is responsible for the execution of more than one LP. As several LPs running on the same processor may simultaneously have at least one non-executed event, a scheduling algorithm must be invoked in order to select the next LP for execution.

Nowadays, the most widely adopted scheduling algorithm, namely Lowest-Timestamp-First (LTF) [8], gives higher priority to LPs having non-executed events with lower timestamps. In particular, from among the LPs having at least one non-executed event, LTF chooses for execution the LP having the non-executed event with the lowest timestamp. LTF implicitly assumes that the event with the lowest timestamp has the lowest probability to be rolled back in the future of the simulation execution as it is the closest one to the Global Virtual Time (i.e. the commitment horizon) of the simulation. If this presumption reveals true, then the amount of rollback of the simulation is actually kept low. Another scheduling algorithm, namely Lowest-Local-Virtual-Time-First (LLVTF) [13], gives higher priority to LPs having lower simulation clocks. In particular, LLVTF chooses for execution the LP with the lowest LVT value and having at least one non-executed event. As the LVT of the LP moves up to the event timestamp upon the execution, the objective of this scheduling algorithm is to prevent any LP from remaining back in simulation time; this will possibly reduce the probability for any LP to induce a timestamp order violation on any other LP. A rather different solution, namely Adaptive Control based scheduling (AC), has been presented in [12]. In this solution, statistics on the past behavior of an LP are collected to establish the “useful work” of the LP (basically the useful work of the LP is evaluated as the ratio between the number of committed events for that LP and the CPU time used by the LP). Higher priority is assigned to LPs with higher useful work values. In [16] a Service Oriented scheduling algorithm (SO) is presented, which relies on the capability of the LPs to predict the timestamps of outgoing simulation messages resulting from the execution of future events. This algorithm gives the highest priority to the LP for which the execution of the next event will produce the simulation message with the minimum predicted timestamp; this is done in order to promptly deliver this message to the recipient LP. Finally a Probabilistic scheduling algorithm (P), has been presented in [18]. In this solution each LP is required to estimate the probability for its next event to be eventually rolled back if currently scheduled for the execution. The event of the LP associated with the lowest estimated probability value is scheduled for the execution. This solution will produce low amount of rollback each time the estimated probability values are good approximations of the real ones; this should happen in simulations with regular patterns for the arrival process of the events in the simulation time.

These algorithms base the scheduling decision exclusively on the comparison of local state information related to the LPs sharing the processor (i.e. LTF relies on the comparison of the timestamps of the next events of the LPs; LLVTF relies on the comparison of the current LVT values; AC relies on the comparison of useful work values; SO relies on the comparison of predicted timestamps; P relies on the comparison of estimated probability values). They do not exploit information related to the current state of the LPs in the (immediate) predecessor set of a given LP. We believe this information is relevant for producing a schedule originating low amount of rollback. The recent theory on near-perfect-state-information (NPSI) synchronization algorithms [21] has shown how this state information plays a fundamental role in the determination of the amount of delay (throttling) after which an event already scheduled for the execution can be actually executed with low probability to be eventually rolled back. In this paper we use such state information for a purpose different from throttling.

Finally, we note that basing a scheduling decision in Time Warp synchronization on non-definitive lower bound values is an action which has some resemblances to the action of determining which events are safe to process in classical conservative synchronization protocols since latter action relies exactly on lower bounds on timestamps which will be carried by future messages. This means that, in some sense, we are exploiting and readapting basic ideas of conservative synchronization concerning the detection of safe events in order to let optimistic (Time Warp) synchronization schedule with higher priority events having high probability to be safe ones. One of the main differences is that conservative protocols use channel information and require additional constraints such as non-decreasing timestamp sequences on a channel; instead our solution uses information on the state of the LPs which is not directly related to properties of the timestamp sequences on any channel.

3 Non-Definitive Lower Bounds

This section enters the description of non-definitive lower bounds, how to compute them and discusses their...
relevance to the problem of rollback generation. Then the adopted piggybacking technique to disseminate lower bound values among the LPs and the data structures to keep track of those values are presented. Before entering such descriptions, let us introduce some simple assumptions and notations. We assume the simulation involves a set of \( n \) LPs, namely \( \{LP_1, \ldots, LP_n\} \). We denote as \( lvt_i \) the current value of the LVT of \( LP_i \), and as \( \delta_i \) the lookahead value for \( LP_i \). The lookahead we consider is explicit (i.e. the lookahead value is known before the simulation begins and does not change during the simulation). We assume, without loss of generality, that for each \( LP_i \) the initial value of \( lvt_i \) is zero (i.e. all LPs initially set their simulation clocks to zero). Communication channels between LPs are assumed to be FIFO (this is not a limitation because the most widely used message passing layers, such as PVM or MPI, actually provide the FIFO property); the relevance of this assumption will be discussed while describing the piggybacking technique. Finally, we denote the immediate predecessor set of \( LP_i \) as \( IP(S(LP_i)) \).

In the context of Time Warp synchronization, we can think of the explicit lookahead notion as the potential for \( LP_i \) to determine the minimum value of the timestamp of events that it will produce in the future of the simulation execution conditional that: (i) \( LP_i \) has reached a given value \( T \) and (ii) \( LP_i \) will push \( lvt_i \) to a value lower than \( T \). We refer to that minimum value, which is a function of the current value of \( lvt_i \) and of the lookahead value \( \delta_i \), as the non-definitive lower bound associated with \( LP_i \). It is non-definitive as it may be revoked in the future of the simulation execution due to a violation of constraint in point (ii). \( LP_i \) keeps track of its current non-definitive lower bound value into a variable, namely \( lb_i \), which is updated as follows:

1. at the beginning of the simulation execution: \( lb_i \leftarrow \delta_i \);
2. anytime \( LP_i \) updates \( lvt_i \): \( lb_i \leftarrow (lvt_i + \delta_i) \).

Concerning point 2., we recall that \( lvt_i \) is updated either when a new event is executed by \( LP_i \) or when \( LP_i \) rolls back. In the first case, the updating leads to an increase of \( lvt_i \), in the second it leads to a decrease of \( lvt_i \). If \( LP_i \) never rolls back, then \( lvt_i \) does never decrease, thus \( lb_i \) does never decrease as well. On the contrary, when a rollback occurs, \( lvt_i \) decreases producing a decrease of \( lb_i \).

We discuss the relevance of the notion of non-definitive lower bound, with respect to the problem of rollback generation, through the following example. Let us consider a simulation involving, among others, three LPs, namely \( LP_1 \), \( LP_2 \) and \( LP_3 \). Suppose \( LP_1 \) and \( LP_2 \) are the only LPs belonging to \( IP(S(LP_3)) \). Suppose \( lb_3 = 30 \) and \( lb_2 = 35 \), there is only one non-executed event, namely \( e_1 \) in the pending-event-queue of \( LP_3 \) and no simulation message addressed to \( LP_3 \) is in transit. The following two cases are possible:

**Case (a).** The timestamp of \( e_1 \) is less than, or equal to, \( \min\{lb_1, lb_2\} \), say \( timestamp(e_1) = 29 \). Suppose \( LP_3 \) executes this event, thus moving \( lvt_k \) to the value 29. In this case, the only way for \( LP_2 \) to rollback and undo \( e_1 \) is: (i) to receive an antievent from either \( LP_1 \) or \( LP_2 \) corresponding to an already executed event of \( LP_3 \), and/or (ii) to receive an event \( e_2 \) with \( timestamp(e_2) < 29 \) from either \( LP_1 \) or \( LP_2 \). Both cases require that either \( LP_1 \) or \( LP_2 \) (or at worst both of them) rolls back to a simulation time \( t \) such that \( t + \delta_i < 29 \) or \( t + \delta_j < 29 \).

**Case (b).** The timestamp of \( e_1 \) is larger than \( \min\{lb_1, lb_2\} \), say \( timestamp(e_1) = 38 \). Suppose \( LP_k \) executes \( e_1 \), thus moving \( lvt_k \) to the value 38. In this case, \( LP_3 \) rolls back undoing \( e_1 \): (i) it receives an antievent from either \( LP_1 \) or \( LP_2 \) corresponding to an already executed event, and/or (ii) it receives an event \( e_2 \) with \( timestamp(e_2) < 38 \) from either \( LP_1 \) or \( LP_2 \). Case (i) requires that either \( LP_1 \) or \( LP_2 \) (or at worst both of them) rolls back to a time \( t \) such that \( t + \delta_i < 30 \) or \( t + \delta_j < 35 \). Instead, case (ii) may occur even if neither \( LP_1 \) nor \( LP_2 \) rolls back. For example, it occurs if \( LP_1 \) produces for \( LP_k \) an event \( e_2 \) with \( timestamp(e_2) = 37 \) (this does not require \( LP_1 \) to roll back).

From previous discussion, it comes out that events of \( LP_3 \) having timestamp less than or equal to \( \min\{lb_1, lb_2\} \) (case a) can be rolled back if, and only if, a timestamp order violation occurs on a process in \( IP(S(LP_3)) \). This is not the case for events having timestamp larger than \( \min\{lb_1, lb_2\} \) (case b). As a reasonable consequence we get that a non-executed event falling in case (a) is less likely to be eventually rolled back (if executed) than a non-executed event falling in case (b) since it can be rolled back only conditional a rollback occurs on another LP. The idea behind the state-based scheduling algorithm presented in Section 4 is to track non-executed events like the one falling in case (a) and to give higher priority to the LPs that must execute them. By previous arguments, this will possibly reduce the amount of rollback of the simulation. Such a tracking needs a system to spread the knowledge on non-definitive lower bound values among the LPs at low cost. As already pointed out, we use a classical piggybacking technique that, in our context, works as follows. Whenever a simulation message \( m \) is sent by \( LP_i \) to \( LP_j \), the current value of \( lb_i \) is piggybacked on \( m \), denoted \( m.lb \). The FIFO property of communication channels guarantees that when \( LP_j \) receives \( m \), no simulation message sent by \( LP_i \) to \( LP_j \) prior to the sending of \( m \) is still in transit. This is an important property as it ensures that no simulation message sent by \( LP_i \) and having timestamp less than \( m.lb \) may be received by \( LP_j \) in the future unless \( LP_i \) actually rolls back after having sent \( m \) to \( LP_j \). If the FIFO property is not guaranteed, it may happen that the non-definitive lower bound \( m.lb \) of \( LP_i \) seen by \( LP_j \) upon the receipt of the simulation message \( m \) may be invalidated not due to a rollback of \( LP_i \) but due to a long transmission delay of a simulation message \( m' \), with timestamp less than \( m.lb \), sent by \( LP_i \) to \( LP_j \) before the sending of \( m \). An example of this is shown in Figure 1. \( LP_i \) sends to \( LP_j \) a simulation message \( m' \) carrying an event with timestamp 20; it then sends a simulation message \( m \) carrying the non-definitive lower bound \( m.lb = 25 \). As \( m \) is delivered before \( m' \), \( LP_j \) interprets that \( LP_i \) will not send simulation messages with timestamp less than 25 unless it rolls back. This reveals wrong upon the receipt of \( m' \) as the lower bound seen by \( LP_j \) is invalidated with no rollback of \( LP_i \).

\( LP_i \) keeps track of the non-definitive lower bound val-


ues of the LPs belonging to $IPS(LP_i)$ by means of a vector $LB_i$ of size $|IPS(LP_i)|$. We denote as $LB_i(LP_j)$ the entry of the vector associated with the logical process $LP_j \in IPS(LP_i)$. $LB_i(LP_j)$ records the latest version of the non-definitive lower bound value of $LP_j$ that $LP_i$ sees. $LB_i$ is updated according to the following rules:

1. at the beginning of the simulation execution: $\forall LP_j \in IPS(LP_i)$, $LB_i(LP_j) \leftarrow \delta_j$;
2. when a simulation message $m$ sent by $LP_j$ is received by $LP_i$: $LB_i(LP_j) \leftarrow m.lb$.

As the LVT of all LPs is set to zero at the beginning of the simulation execution, rule 1 trivially states that $LP_j \in IPS(LP_i)$ never will produce for $LP_i$ any simulation message with timestamp which is less than the lookahead $\delta_j$. Rule 2., together with the FIFO property of communication channels, state that each time $LP_j$ receives a new simulation message $m$ from $LP_j$ the younger version of $lb_j$ carried by $m$ replaces the existing one. Recall that a simulation message carries either an event or an antievnet, therefore the communication overhead due to rollback (i.e. messages carrying antievents) is not necessarily only wasted effort since it helps to spread the state information useful for the scheduling decisions. Furthermore, note that when a simulation message $m$ is received by $LP_i$, the value $m.lb$ is extracted and the vector $LB_i$ is updated independently from how far in time the event/antievnet carried by $m$ will be processed.

Previous scheme to spread and store non-definitive lower bound values induces negligible additional communication and/or computational overhead. In particular, the size of the data piggybacked on any simulation message is small (usually 8 bytes if non-definitive lower bound values are represented as doubles), no additional message must be exchanged, and the processing time needed to update the entries of the vector is negligible.

4 State-Based Scheduling

In this section we present the state-based scheduling algorithm relying on the piggybacking technique described in Section 3. The key problem of this technique is that due to the communication latency and/or, more important, to the fact that $LP_j$ might send simulation messages to $LP_i$ infrequently, the value $LB_i(LP_j)$ at a given real time instant is actually only an approximation of the current value of $lb_j$ at that real time instant. There exists the possibility that recorded information about non-definitive lower bound values becomes stale in the course of the simulation execution. Therefore, state information must be filtered in order to discard stale values. In the next section we tackle the information filtering problem and show how to keep track of non-stale information at low cost. In the same section we introduce the notion of filter length defined as the amount of non-stale information. Then we describe the scheduling algorithm which uses only the non-stale information. Finally we present the iterative procedure for the calculation of an adequate value for the filter length which maximizes the speed of the simulation execution.

4.1 Information Filtering

Let us associate with each entry $LB_i(LP_j)$ of the vector $LB_i$ an age, which is defined as the elapsed real time since $LB_i(LP_j)$ was last updated. No two entries of the vector $LB_i$ are updated at the same real time instant by $LP_i$, therefore we can build at any time a total ordering on the entries of the $LB_i$ vector based on increasing values of their age. The first entry in the ordering corresponds to the entry with the minimum age; the last element in the ordering corresponds to the entry with the maximum age. The ordering changes during the simulation execution due to updates issued on the vector entries. Anytime an entry is updated it becomes the first element in the ordering as its age becomes the minimum one.

For each logical process $LP_i$ we define the filtered state information as the set of the $fl_i$ values (with $0 \leq fl_i \leq |IPS(LP_i)|$) of the most recently updated entries of the vector $LB_i$. Hence, the variable $fl_i$ represents the filter length associated with $LP_i$. $LP_i$ can keep track of the filtered state information by maintaining a linked list $L_i$. Any element in the list is a pointer to an entry of the $LB_i$ vector. The list is updated as follows:

1. at the beginning of the simulation execution the list is initialized as empty;
2. anytime the entry $LB_i(LP_j)$ of the vector $LB_i$ is updated (i.e. $LP_i$ receives a simulation message from $LP_j$) then a pointer $p$ to $LB_i(LP_j)$ is inserted at the head of the list; if a pointer $p' \neq LB_i(LP_j)$ was already in the list, then $p'$ is removed from the list; if after the insertion of the pointer $p$ the size of $L_i$ exceeds the filter length $fl_i$ then the pointer at the tail of the list is removed.

The elements in the list $L_i$ point, at any time, to the $fl_i$ most recently updated values of the non-definitive lower bounds of LPs in the immediate predecessor set of $LP_i$. Note that previous solution keeps track of these $fl_i$ most recently updated values without the need for computing age values explicitly, which could induce relevant overhead ($\frac{3}{4}$).

The time complexity for the operation in point 2. can be rendered $O(1)$ by simply augmenting the entries of the $LB_i$ vector with pointers to the corresponding elements of the $L_i$ list, linking the elements of $L_i$ in both directions and maintaining a pointer to the element at the tail of the list.

If age values must be computed explicitly, then anytime an update is issued on an entry of the vector, an access to the clock value has to be performed. In some case this might result expensive.
4.2 The Scheduling Algorithm

We suppose there exists a scheduler module that has access to the pending-event-queue of any LP hosted on the processor, therefore, for any logical process \(LP_i\), it knows the timestamp value \(t_{s_i}\) of the event that has the minimum timestamp among the non-executed events of \(LP_i\), if any. This is not a limitation as most previous schedulers [8, 16, 18] must have access exactly to the same information in order to perform the scheduling decision. If there is no pending event in the pending-event-queue of \(LP_i\), then the value assigned to \(t_{s_i}\) is \(+\infty\). In addition, we suppose the scheduler module has access to the filtered information associated with any list \(L_i\) and, also, it can manipulate the filter length \(f_l\) of \(LP_i\).

The scheduler module keeps an integer variable \(SFL\) (Scheduling-Filter-Length) which records the maximum value allowed for the filter of any LP hosted on the processor. Specifically, for each \(LP_i\) hosted on the processor, \(f_l = \min(SFL, \left| IPS(LP_i)\right|)\). Basically, \(SFL\) records the maximum number of elements any logical process \(LP_i\) is allowed to keep in its list \(L_i\). If the value of \(SFL\) is larger than the cardinality \(\left| IPS(LP_i)\right|\) of the immediate predecessor set of \(LP_i\), then \(LP_i\) is allowed to keep in \(L_i\) the pointers to all the entries of the \(LB_i\) vector.

Our state-based scheduling algorithm relies on the partitioning of the LPs in two distinct priority classes: high and low. It gives high priority to any \(LP_i\) such that \(t_{s_i}\) is less than the minimum lower bound value pointed by the elements of the list \(L_i\). In other words, the algorithm gives high priority to any \(LP_i\) having the next event with timestamp less than the minimum from among the lower bounds belonging to the filtered information associated with \(LP_i\). The reason for this is as follows. The filtered information associated with the list of pointers \(L_i\) represents the most recent state information seen by \(LP_i\) about the non-definitive lower bound values of the LPs in its immediate predecessor set. If \(LP_j \in IPS(LP_i)\) but \(LB_j(LP_j)\) does not belong to the filtered information, then \(LP_j\) does not send simulation messages to \(LP_i\) from long time. As a reasonable consequence it is extremely likely that the simulation clock of \(LP_j\) has moved up so the current value of \(LB_j\) has moved up as well. Therefore it is likely that the minimum lower bound value associated with the list \(L_i\) actually corresponds to the minimum simulation time from which the LPs belonging to \(IPS(LP_i)\) may actually send simulation messages to \(LP_i\) unless one of them is involved in a rollback. In Figure 2 the scheduling algorithm is shown.

The time complexity for the definition of the set \(\beta\) (containing the high priority LPs) in line 1 is a linear function of the number of elements of the \(L_i\) lists (or at best a logarithmic function if additional complex data structures to maintain the minimum of the filtered state information are added). This number is strongly related to the Scheduling-Filter-Length \(SFL\) kept by the scheduler. This points out that \(SFL\) is a relevant parameter that determines the delay of the scheduling decision. If \(SFL\) is small then the delay is short, therefore the simulation can take advantage of the implicit throttling originated by the scheduling algorithm \(^3\). If the value of \(SFL\) is large the simulation may suffer from excessive delay of event execution due to a long delay of the scheduling decision. In addition we note that \(SFL\) has strong impact also on the quality of the filtered state information. Specifically, too small values for \(SFL\) may push the LPs to discard non-stale information from the filtered one which could be useful for the scheduling decision. If the value of \(SFL\) is too large, the filtered information may contain stale values. The problem of the selection of suitable values for \(SFL\) in order to optimize the gain from throttling and scheduling based on high quality filtered information against the loss from scheduling based on poor quality filtered information and excessive event execution delay will be tackled in the next section.

4.3 Selection of the Scheduling-Filter-Length

The procedure we propose for the selection of an adequate value of \(SFL\) is iterative and relies on the monitoring of the event rate (i.e. committed events per second). This parameter indicates how fast the simulation progresses. Before entering the description of the procedure we note that the event rate which should be optimized through the state-based scheduling decision is the whole event rate of the simulation (optimization of the event rate on a single processor or for a single LP does not necessarily lead to performance improvements). For this reason the scheduler modules on distinct processors must keep the same value for \(SFL\) in order to monitor the real impact of its variations on the event rate of the simulation.

We consider the execution of the simulation as partitioned into observation periods. Guidelines to select the length of periods ensuring that statistical data collected in any of them are meaningful have been discussed in [17]. We assume there exists a master processor which gathers statistical data related to the event rate collected on the remote processors and performs the iterative procedure for tuning the value of \(SFL\). Each time a new value for \(SFL\) is computed, it is notified to the remote processors. Specifically, the interaction between master and slave processors is as follows. When the observation period expires at the master processor, it sends a request message to the slave processors. Upon the receipt of this message, any slave processor replies with a message whose payload is the locally observed event rate since the last request was processed. When

\(^3\)The reader can refer to [2, 3, 21] for information on how “adequately short” event execution delay influences positively the rollback of the simulation.
all the replies are received by the master processor, it computes the observed event rate of the simulation in the current observation period by computing the sum of the event rates on all the processors. It then selects the new value for $SFL$ and notifies it to the slave processors. Such an interaction takes place until the iterative procedure to adjust $SFL$ ends. Note that the simulation is not frozen while the interaction between processors takes place.

The iterative procedure below described for the selection of the value of $SFL$ exploits the property that if $SFL$ is set to zero then any list $L_i$ is always empty so the state-based scheduling algorithm boils down to the classical LTF algorithm as no LP will ever belong to the set $\beta$ of the high priority LPs (line 4 of the algorithm is never executed). Let us now enter the description of the procedure. At the beginning of the simulation execution $SFL$ is set to zero by the master processor and this value is notified to the slave ones. This means that during the first period the scheduler module on each processor behaves exactly like LTF. At the end of the successive observation periods, the master processor increases the value of $SFL$ by one if the observed event rate of the simulation did not decrease. If later quantity decreases, the master processor decreases $SFL$ by one, notifies it to the slave processors and the procedure ends (i.e. no further modification of $SFL$ is performed).

This iterative procedure aims at increasing the value of $SFL$ in order to induce the scheduler module to behave differently from LTF by keeping into account state information in the scheduling decision. If the scheduling decisions based on the state information are effective, i.e. the event rate grows compared to that of LTF observed in the first period, then the value of $SFL$ is further increased in order to keep into account more state information, although less recently updated. The growth of $SFL$ stops when the decrease of the event rate is detected. As discussed in the previous section, the final value selected by the procedure should balance the gain/loss from throttling and the gain/loss from scheduling relying on good/poor quality state information. As upper limit on $SFL$ we can select the maximum cardinality of the immediate predecessor sets of the LPs. If such value is reached through the iterative procedure, then it means any state information is always useful (i.e. no state information on non-definitive lower bounds becomes ever stale for the scheduling decision).

The termination condition of the procedure (i.e. the decrease of the observed event rate), together with the consequent decrease of $SFL$ by one, allow us to find a local maximum for the event rate. Therefore the procedure does not guarantee that the optimal value of the event rate as a function of $SFL$ is reached. In this aspect, our solution resembles the iterative procedure presented in [9] for the selection of the checkpoint interval in Time Warp simulations. As a final consideration, we note that the iterative procedure may be run periodically in order to prevent the last selected value for $SFL$ from becoming inadequate due to variations of the behavior of the LPs or of the load on the machines.

5 Performance Results

In this section experimental results are reported to compare the performance achievable by using the state-based scheduling algorithm proposed in this paper (hereafter SB) to the one of the classical LTF algorithm. As hardware architecture we used a cluster of 4 Pentium II 300 MHz - 128 Mbytes RAM, connected via fast switched Ethernet. Interprocessor communication relies on message passing supported by PVM [23]. In our simulator antievents are sent aggressively (i.e. as soon as the LP rolls back [6]) and fossil collection is executed periodically. We tested the performance of the scheduling algorithms using the synthetic benchmark known as PHOLD model, originally presented in [5], which consists of a fixed number of LPs and of a constant number of jobs circulating among the LPs (which is referred to as job population). In this benchmark, the routing of jobs among the LPs and the timestamp increments are taken from some stochastic distributions. We recall that this benchmark is one of the most used for testing performance of scheduling algorithms [16, 18], checkpointing techniques [14, 15, 20, 25] and also throttling schemes [21]. Its popularity relies in that it shows a rollback behavior similar to many other synthetic benchmarks and to several real world models. We considered a PHOLD model consisting of 64 homogeneous LPs, with a job population of 10 jobs per LP. Jobs are equally likely to be forwarded to any other LP, so the immediate predecessor set of any LP consists of 63 elements. In our experiments the timestamp increment is originated as the sum of two quantities: LAH and $R$. The first quantity, namely $LAH$, is deterministic and does not change in the lifetime of the simulation. It represents the lookahead of the LPs. The second quantity, namely $R$, is a random value obtained by an exponential distribution with mean 10 simulation time units. We varied $LAH$ from 0 to 4, thus obtaining 5 different configurations concerning the timestamp increments. Low values for $LAH$ originate simulations with poor lookahead features. Instead, high values originate simulations with very good lookahead. Any event (i.e. any job) takes around 200 microseconds to process; as checkpointing protocol we adopted copy state saving (i.e. the state of an LP is saved before the execution of any new event) and the state saving time is fixed at around 100 microseconds. In our experiments, each machine runs the same number of LPs (no other user load runs on any machine). We report measures related to the following parameters:

- the efficiency ($EFF$), that is the ratio between the number of committed events and the total number of executed events (committed plus rolled back); this parameter indicates the probability for any event to be eventually committed once executed, it is therefore representative of the impact of the scheduling algorithm on the amount of rollback of the simulation;
- the event rate ($ER$), i.e. committed events per second; this parameter indicates the impact of the scheduling algorithm on the speed of the simulation execution; it is therefore representative of the achieved performance;
- the observed probability, namely $D$, that SB schedules for execution an LP distinct from that which would be scheduled by LTF considering the same situation for the pending-event-queues of the LPs; this param-
eter allows us to point out whether the possible performance difference between SB and LTF is actually originated by different outcomes of the scheduling decision or by delays introduced by the higher time complexity of SB compared to LTF;

- the value of $SFL$ selected by the iterative procedure described in Section 4.3.

For each configuration of the benchmark we report the average observed values of previous parameters, computed over 10 runs that were all done with different seeds for the random number generation. At least $2 \times 10^6$ committed events were simulated in each run. The results are reported in Figures from 3 to 6.

![Figure 3. Efficiency vs LAH.](image1)

![Figure 4. Event Rate vs LAH.](image2)

![Figure 5. D vs LAH.](image3)

![Figure 6. SFL vs LAH.](image4)

Plots in Figure 3 and in Figure 4 show that for all the considered values of LAH both $EFF$ and $ER$ are increased when using SB. Therefore SB reduces the amount of rollback of the simulation and this reduction actually leads to faster execution of the simulation.

From the observed values of $D$ in Figure 5 we have that when $LAH = 0$ (i.e. when the simulation exhibits no lookahead property) then SB performs almost always the same scheduling decision that would be performed by LTF.

For $SFL$ we report the value achieved by running the iterative procedure one time. However we noted that running the procedure more that one time during the simulation execution leads, in most of the cases, to the same final value for $SFL$. This points out that our simulations are likely to reach a steady state.

when considering the same situation for the pending-event-queues of the LPs. In particular, the probability $D$ of different scheduling decision is around 4%. Instead, when the simulation has at least minimum lookahead properties (i.e. $LAH \geq 1$), the probability $D$ to perform different scheduling decisions quickly grows and gets a stable value which is around 20%. From plots in Figure 6 we have that when $LAH = 0$, the average observed value for $SFL$ is around 9; instead, for higher values of $LAH$, $SFL$ quickly decreases and gets a stable value which is around 5.

The combination of the plots in Figure 5 and in Figure 6 points out that for the case of $LAH = 0$ the performance gain achieved by SB is not due to real different scheduling decisions compared to LTF; instead it is due to the throttling effect produced by the higher time complexity of SB compared to LTF. Recall that when $LAH = 0$, $SFL$ gets its maximum value, thus originating the maximum delay for the scheduling decision of SB compared to LTF. As soon as $LAH$ gets a value larger than zero, the value of $SFL$ quickly decreases, thus the throttling effect tends to disappear. In this case the performance gain of SB compared to LTF is due also to real different scheduling decisions.

As shown by plots in Figure 4, the performance gain of SB is higher (around 9%-10%) for low values of $LAH$ (i.e. when $LAH \leq 3$). This behavior is an expected one when considering that for high values of $LAH$ the simulation exhibits high efficiency (i.e. low amount of rollback) also under LTF, thus reducing the margin of improvement achievable through a more sophisticated scheduling algorithm like SB. Another interesting point is in that for all the configura-
tions the iterative procedure for the selection of $SFL$ leads to values different from zero. This points out that for the considered simulation model the first maximum of the event rate as a function of $SFL$ is actually found when at least a minimum amount of state information is taken into account in the scheduling decision.

6 Conclusions and Future Work

In this paper we have presented a scheduling algorithm for the selection of the next LP to be run on a processor in Time Warp synchronized parallel discrete event simulators. In our algorithm the priority of the LPs depends on both the timestamp of their next event and state information related to the LPs in their immediate predecessor set. In this aspect our algorithm differs from all existing algorithms. We have presented a solution for collecting state information at low cost and also for manipulating it in order to discard stale values. Latter action relies on an iterative procedure for the selection of the length of a filter for the state information. We have tested the algorithm using a classical benchmark in different configurations and we have observed a reduction of the amount of rollback of the simulation, compared to the classical Lowest-Timestamp-First algorithm, which leads to a real increase of the speed of the simulation execution.

The presented results are encouraging and point out the relevance of the use of state information related to the LPs in the immediate predecessor sets for the scheduling decision. However a number of issues remain still open. As an example, one of the critical points of our technique relies in that the iterative procedure for the selection of the filter length stops when the first maximum for the monitored performance parameter (i.e. the event rate) is observed. This solution might reveal ineffective anytime the first maximum is observed for the initial value of the filter length that is zero in our solution. In this case the algorithm behaves exactly like Lowest-Timestamp-First. We are currently carrying out a simulation study on models different from the one studied in Section 5 and we have noted that in some cases no performance gain is achieved, compared to Lowest-Timestamp-First, exactly due to the final outcome of the iterative procedure for the selection of the filter length. Immediate future work includes the definition of procedures for the selection of the filter length which should overcome this problem.

References