Parallel and Distributed Simulation of Discrete Event Systems

Alois Ferscha*  
Institut für Angewandte Informatik  
Universität Wien  
Lenaugasse 2/8, A-1080 Vienna, AUSTRIA  
Email: ferscha@ani.univie.ac.at

Satish K. Tripathi  
Computer Science Department  
University of Maryland  
College Park, MD 20742, U.S.A.  
Email: tripathi@cs.umd.edu

Abstract

The achievements attained in accelerating the simulation of the dynamics of complex discrete event systems using parallel or distributed multiprocessing environments are comprehensively presented. While parallel discrete event simulation (DES) governs the evolution of the system over simulated time in an iterative SIMD way, distributed DES tries to spatially decompose the event structure underlying the system, and executes event occurrences in spatial subregions by logical processes (LPs) usually assigned to different (physical) processing elements. Synchronization protocols are necessary in this approach to avoid timing inconsistencies and to guarantee the preservation of event causalities across LPs.

Included in the survey are discussions on the sources and levels of parallelism, synchronous vs. asynchronous simulation and principles of LP simulation. In the context of conservative LP simulation (Chandy/Misra/Bryant) deadlock avoidance and deadlock detection/recovery strategies, Conservative Time Windows and the Carrier Nullmessage protocol are presented. Related to optimistic LP simulation (Time Warp), Optimistic Time Windows, memory management, GVT computation, probabilistic optimism control and adaptive schemes are investigated.


General Terms: Algorithms, Performance

Additional Key Words and Phrases: Parallel Simulation, Distributed Simulation, Conservative Simulation, Optimistic Simulation, Synchronization Protocols, Memory Management

---

*This work was conducted while Alois Ferscha was visiting the University of Maryland, Computer Science Department, supported by a grant from the Academic Senate of the University of Vienna.
1 Introduction

Modeling and analysis of the time behavior of dynamic systems is of wide interest in various fields of science and engineering. Common to ‘realistic’ models of time dynamic systems is their complexity, very often prohibiting numerical or analytical evaluation. Consequently, for those cases, simulation remains the only tractable evaluation methodology. Conducting simulation experiments is, however, time consuming for several reasons. First, the design of sufficiently detailed models requires in-depth modeling skills and usually extensive model development efforts. The availability of sophisticated modeling tools today significantly reduces development time by standardized model libraries and user-friendly interfaces. Second, once a simulation model is specified, the simulation run can take exceedingly long to execute. This is due either to the objective of the simulation, or the nature of the simulated model. For statistical reasons it might for example be necessary to perform a whole series of simulation runs to establish the required confidence in the performance parameters obtained by the simulation, or in other words make confidence intervals sufficiently small. Another natural consequence why simulation should be as fast as possible comes from the objective of exploring large parameter spaces, or to iteratively improve a parameter estimate in a loop of simulation runs. The simulation model as such might require tremendous computational resources, making the use of contemporary 100 MFLOPs computers hopeless.

Possibilities to resolve these shortcomings can be found in several methods, one of which is the use of statistical knowledge to prune the number of required simulation runs. Statistical methods like variance reduction can be used to avoid the generation of “unnecessary” system evolutions, in the sense that statistical significance can be preserved with a smaller number of evolutions given the variance of a single random estimate can be reduced. Importance sampling methods can be effective in reducing computational efforts as well. Naturally, however, faster simulations can be obtained by using more computational resources, particularly multiple processors operating in parallel. It seems obvious at least for simulation models reflecting real life systems constituted by components operating in parallel, that this inherent model parallelism could be exploited to make the use of a parallel computer potentially effective. Moreover, for the execution of independent replications of the same simulation model with different parametrizations the parallelization appears to be trivial. In this work we shall systematically describe ways of accelerating simulations using multiprocessor systems with focus on the synchronization of logical simulation processes executing in parallel on different processing nodes in a parallel or distributed environment.

1.1 Continuous vs. Discrete Event Simulation

Basically every simulation model is a specification of a physical system (or at least some of its components) in terms of a set of states and events. Performing a simulation thus means mimicking the occurrence of events as they evolve in time and recognizing their effects as represented by states. Future event occurrences induced by states have to be planned (scheduled). In a continuous simulation, state changes occur continuously in time, while in a discrete simulation the occurrence of an event is instantaneous and fixed to a selected point in time. Because of the convertability of continuous simulation models into discrete models by just considering the start instant as well as the end instant of the event occurrence, we shall subsequently only consider discrete simulation.
1.2 Time Driven vs. Event Driven Simulation

Two kinds of discrete simulation have emerged that can be distinguished with respect to the way simulation time is progressed. In time driven discrete simulation simulated time is advanced in time steps (or ticks) of constant size $\Delta$, or in other words the, observation of the simulated dynamic system is discretized by unitary time intervals. The choice of $\Delta$ interchanges simulation accuracy and elapsed simulation time: ticks short enough to guarantee the required precision generally imply longer simulation time. Intuitively, for event structures irregularly dispersed over time, the time-driven concept generates inefficient simulation algorithms.

Event driven discrete simulation discretizes the observation of the simulated system at event occurrence instants. We shall refer to this kind of simulation as discrete event simulation (DES) subsequentially. A DES, when executed sequentially repeatedly processes the occurrence of events in simulated time (often called “virtual time”, VT) by maintaining a time ordered event list (EVL) holding timestamped events scheduled to occur in the future, a (global) clock indicating the current time and state variables $S = (s_1, s_2, \ldots, s_n)$ defining the current state of the system (see Figure 1). A simulation engine (SE) drives the simulation by continuously taking the first event out of the event list (i.e. the one with the lowest timestamp), simulating the effect of the event by changing the state variables and/or scheduling new events in EVL – possibly also removing obsolete events. This is performed until some pre-defined endtime is reached, or there are no further events to occur.

As an example, assume a physical system of two machines participating in a manufacturing process. In a preprocessing step, one machine produces two subparts A1 and A2 of a product A, both of which can be assembled concurrently. Part A1 requires a single, whereas A2 takes a nonpredictable amount of assembly steps. Once one A1 and one A2 are assembled (irrespective of the machine that assembled it) one piece of a A is produced.

The system is modelled in terms of a Petri net (PN): transition $t_1$ models the preprocessing step and the forking of independent post-processing steps, $t_2$ and $t_3$. Machines in the preprocessing phase are represented by tokens in $p_1$, finished parts A1 by tokens in $p_5$ and finished or “still in assembly” parts A2 by tokens in $p_4$. Once there is at least one token in $p_5$ and at least one token in $p_4$, the assembly process stops or repeats with equal probability (conflict among $t_4$ and $t_5$). Once the assembly process terminates yielding one A, one machine is released ($t_5$). The time behavior of the physical system is modelled by associating timing information to transitions ($\tau(t_1) = 3$, $\tau(t_2) = \tau(t_3) = 2$ and $\tau(t_4) = \tau(t_5) = 0$). This means that a transition $t_i$ that became enabled by the arrival of tokens in the input places at time $t$ and remained enabled (by the presence of tokens in the input places) during $[t, t + \tau(t_i)]$. It fires at time $t + \tau(t_i)$ by removing tokens from input
places and depositing tokens in $t_1$'s output places. The initial state of the system is represented by the marking of the PN where place $p_1$ has 2 tokens (for 2 machines), and no tokens are available in any other place. Both the time driven and the event driven DES of the PN are illustrated in Figure 2.

The time driven DES increments VT (denoted by a watch symbol in the table) by one time unit each step, and collects the state vector $S$ as observed at that time. Due to time resolution and non time consuming state changes of the system, not all the relevant information could be collected with this simulation strategy.

The event driven DES employs a simulation engine as in Figure 1 and exploits a natural correspondence among event occurrences in the physical system and transition firings in the PN model by relating them: whenever an event occurs in the real (physical) system, a transition is fired in the model. The event list hence carries transitions and the time instant at which they will fire, given that the firing is not preempted by the firing of another transition in the meantime. The state of the system is represented by the current PN marking ($S$), which is changed by the processing of an event, i.e. the firing of a transition: the transition with the smallest timestamp is withdrawn from the event list, and $S$ is changed according to the corresponding token moves. The new state, however, can enable new transitions (in some cases maybe even disable enabled transitions), such that EVL has to be corrected accordingly: new enabled transitions are scheduled with their firing time to occur in the future by inserting them into EVL (while disabled transitions are removed). Finally the VT is set to the timestamp (ts) of the transition just fired.

Related now to the example in Figure 2 we have: Before the first step of the simulation starts, VT is set to 0 and transition $t_1$ is scheduled twice for firing at time $0 + \tau(t_1) = 3$ according to the initial state $S = (2, 0, 0, 0, 0)$. There are two identical event entries with identical timestamps in the EVL, announcing two event occurrences at that time. In the first simulation step, one of these events (since both have identical, lowest timestamps) is taken out of EVL arbitrarily, and the state is changed to $S = (1, 1, 1, 0, 0)$ since firing $t_1$ removes one token from $p_1$ and generates one token for both $p_2$ and $p_3$. Finally VT is adjusted. The new marking now enables transitions $t_2$ and $t_3$, both with firing time 2. Hence, new event tuples $(t_2 \oplus VT + \tau(t_2))$ and $(t_3 \oplus VT + \tau(t_3))$ are generated and scheduled, i.e. inserted in EVL in increasing order of timestamp. In step 2, again, the event with smallest timestamp is taken from EVL and processed in the same manner, etc.

1.3 Accelerating Simulations

In the example of Figure 2, there are situations where several transitions have identical smallest timestamps, e.g. in step 5 where all scheduled transitions have identical end firing time instants. This is not an exceptional situation but appears whenever (i) two or more events (potentially) can occur at the same time but are mutually exclusive in their occurrence, or (ii) (actually) do occur simultaneously in the physical system. The latter distinction is very important with respect to the construction of parallel or distributed simulation engines: $t_2$ and $t_3$ are scheduled to fire at time $\delta$ (their enabling lasted for the whole period of their firing time $\tau(t_2) = \tau(t_3) = 2$), where the firing of one of them will not interfere the firing of the other one. $t_2$ and $t_3$ are said to be concurrent events since their occurrences are not interrelated. Obviously $t_2$ and $t_3$ could be simulated in parallel, say $t_2$ by some processor P1 and $t_3$ by another processor P2. As an improvement of the sequential simulation on the other hand, they could both be removed from EVL in a single simulation step. The situation is somewhat different with $t_4$ and $t_5$, since the occurrence of one of them will disable
the other one — $t_4$ and $t_5$ are said to be conflicting events. The effect of simulating one of them would (besides changing the state) also be to remove the other one from EVL. $t_4$ and $t_5$ are mutually exclusive and preclude parallel simulation.

Before following the idea of simulating a single simulation model (like the example PN) in parallel, we will first take a more systematic look at the possibilities to accelerate the execution of simulations using $P$ processors.

### 1.3.1 Levels of Parallelism/Distribution

**Application-Level** The most obvious acceleration of simulation experiments with the aim to explore large search spaces is to assign independent replications of the same simulation model with possibly different input parameters to the available processors. Since no coordination is required between processors during their execution high efficiency can be expected. The sequential simulation code can be reused avoiding costly program parallelization and problem scalability is unlimited. Distributing whole simulation experiments, however, might not be possible due to memory space limitations in the individual processing nodes.

**Subroutine-Level** Simulation studies in which experiments must be sequenced due to iteration dependencies among the replications, i.e., input parameters of replication $i$ are determined by the output values of replication $i-1$, naturally preclude application-level distribution. The distribution of subroutines constituting a simulation experiment, like random number generation, event processing, state update, statistics collection might be effective for acceleration in this case. Due to a rather small amount of simulation engine subtasks, the amount of processors that can be employed, and thus the degree of attainable speedup, is limited with a subroutine-level distribution.

**Component-Level** Neither of the two distribution levels above makes use of the parallelism available in the physical system being modelled. For that, the simulation model has to be decomposed into model components or submodels, such that the decomposition directly reflects the inherent model parallelism or at least preserves the chance to gain from it during the simulation.
run. A natural simulation problem decomposition could be the result of an object oriented system
design, where object class instances corresponding to (real) system components represent computa-
tional tasks to be assigned to parallel processors for execution. A queueing network workflow
model of a business organization for example, that directly reflects organizational units like offices
or agents as single queues, defines in a natural way the decomposition and assignment of the sim-
ulation experiment to a multiprocessor. The processing of documents by an agent then could be
simulated by a processor, while the document propagation to another agent in the physical system
could be simulated by sending a message from one processor to the other.

**Event-Level, Centralized EVL**  Model parallelism exploitation at the next lower level aims at
a distribution of single events among processors for their concurrent execution. In a scheme where
EVL is a centralized data structure maintained by a master processor, acceleration can be achieved
by distributing (heavy weighted) concurrent events to a pool of slave processors dedicated to execute
them. The master processors in this case takes care that consistency in the event structure is
preserved, i.e. prohibits the execution of events potentially yielding causality violations due to
overlapping effects of events being concurrently processed. As we have seen with the example in
Figure 2 (step 5 in the event driven simulation), this requires knowledge about the event structure
which must be extracted from the simulation model. The distribution at the event level with a
centralized EVL is particularly appropriate for shared memory multiprocessors where EVL can be
implemented as a shared data structure accessed by all processors. The events processed in parallel
are typically the ones located at the same time moment (or small epoch) of the space-time plane.

**Event-Level, Decentralized EVL**  The most permissive way of conducting simulation in par-
allel is at the level where events from arbitrary points of the space-time are assigned to di-
fferent processors, either in a regular or an unstructured way. Indeed, a higher degree of parallelism can
be expected to be exploitable in strategies that allow the concurrent simulation of events with dif-
ferent timestamps. Schemes following this idea require protocols for local synchronization, which
may in turn cause increased communication costs depending on the event dispersion over space and
time in the underlying simulation model. Such synchronization protocols have been the objective
of parallel and distributed simulation research, which has received significant attention since the
proliferation of massively parallel and distributed computing platforms.

**1.4 Parallel vs. Distributed Simulation**

An important distinction of parallel or multiple processor machines is their operational principle.
In a SIMD operated environment, a set of processors perform identical operations on different data
in lock step. Each processor possesses its own local memory for private data and programs, and
executes an instruction stream controled by a central unit. Though the size of data items might
vary from a simple datum to a complex data set, and although the instruction could be a complex
computer program, the control unit forces synchronism among the independent computations.
Physically, SIMD operated computers have been implemented on shared memory architectures or on
distributed memory architectures with static, regular interconnection networks as a means of data
exchange. Whenever the synchronism imposed by the SIMD operational principle is exploited to
conduct simulation with P processors (under central control) we shall talk about parallel simulation.
An alternative design to SIMD is the MIMD model of parallel computation. A collection of processes as assigned to processors operate asynchronously in parallel, usually employing message passing as a means of communication. In contrast to SIMD, communication in a MIMD operated computer has the purpose of data exchange, but also of locally synchronizing the communicating processes’ activities. The generality of the MIMD model adds another difficulty to the design, implementation and execution of parallel simulations, namely the necessity of an explicit encoding of a synchronization strategy in the parallel simulation program. We shall refer to simulation strategies using P processors with an explicit encoding of synchronization among processes by the term distributed simulation.

1.5 Logical Process Simulation

Common to all simulation strategies with distribution at the event level is their aim to divide a global simulation task into a set of communicating logical processes (LPs), trying to exploit the parallelism inherent among the respective model components with the concurrent execution of these processes. We can thus view a logical process simulation (LP simulation) as the cooperation of an arrangement of interacting LPs, each of them simulating a subspace of the space-time which we will call an event structure region. In our example a region would be a spatial part of the PN topology. Generally a region is represented by the set of all events in a sub-epoch of the simulation time, or the set of all events in a certain subspace of the simulation space.

The basic architecture of an LP simulation can be viewed as in Figure 3:

- A set of LPs is devised to execute event occurrences synchronously or asynchronously in parallel.
- A communication system (CS) provides the possibility to LPs to exchange local data, but also to synchronize local activities.
- Every LP has assigned a region \( R_i \) as part of the simulation model, upon which a simulation engine \( SE_i \) operating in event driven mode (Figure 1) executes local (and generates remote) event occurrences, thus progressing a local clock (local virtual time, LVT).
Each LP $i$ (SE$_i$) has access only to a statically partitioned subset of the state variables $S_i \subset S$, disjoint to state variables assigned to other LPs.

Two kinds of events are processed in each LP $i$: internal events which have causal impact only to $S_i \subset S$, and external events also affect $S_j \subset S$ ($i \neq j$) the local states of other LPs.

A communication interface CI$_i$ attached to the SE takes care for the propagation of effects causal to events to be simulated by remote LPs, and the proper inclusion of causal effects to the local simulation as produced by remote LPs. The main mechanism for this is the sending, receiving and processing of event messages piggybacked with copies of the senders LVT at the sending instant.

Basically two classes of CIs have been studied for LP simulation, either taking a conservative or an optimistic position with respect to the advancement of event executions. Both are based on the sending of messages carrying causality information that has been created by one LP and affects one or more other LPs. On the other hand, the CI is also responsible for preventing global event causality violations. In the first case, the conservative protocol, the CI triggers the SE in a way which prevents from causality errors ever occurring (by blocking the SE if there is the chance to process an ‘unsafe’ event, i.e. one for which causal dependencies are still pending). In the optimistic protocol, the CI triggers the SE to redo the simulation of an event should it detect that premature processing of local events is inconsistent with causality conditions produced by other LPs. In both cases, messages are invoked and collected by the CIs of LPs, the propagation of which consumes real time dependent on the technology the communication system is based on. The practical impact of the CI protocols developed in theory therefore is highly related to the effective technology used in target multiprocessor architectures. (We shall avoid presenting the achievements of research in the light of readily available technology, permanently being subject to change.)

For the representation and advancement of simulated time (VT) in an LP simulation we can devise two possibilities[Peac 79]: a synchronous LP simulation implements VT as a global clock, which is either represented explicitly as a centralized data structure, or implicitly implemented by a time-stepped execution procedure – the key characteristic being that each LP (at any point in real time) faces the same VT. This restriction is relaxed in an asynchronous LP simulation, where every LP maintains a local VT (LVT) with generally different clock values at a given point in real time.

1.5.1 Synchronous LP Simulation

In a time-stepped LP simulation [Peac 79], all the LPs’ local clocks are kept at the same value at every point in real time, i.e., every local clock evolves on a sequence of discrete values $(0, \Delta, 2\Delta, 3\Delta, \ldots)$. In other words, simulation proceeds according to a global clock since all local clocks appear to be just a copy of the global clock value. Every LP must process all events in the time interval $[i\Delta, (i+1)\Delta)$ (time step $i$) before any of the LPs are allowed to begin processing events with occurrence time $(i+1)\Delta$ and after. This strategy considerably simplifies the implementation of correct simulations by avoiding problems of deadlock and possibly overwhelming message traffic and/or memory requirements as will be seen with synchronization protocols for asynchronous simulation. Moreover, it can efficiently use the barrier synchronization mechanisms available in almost every parallel processing environment. The imbalance of work across the LPs in certain time steps on the other hand naturally leads to idle times and thus represents a source of inefficiency.
Both centralized and decentralized approaches of implementing global clocks have been followed. In [Venk 86], a centralized implementation with one dedicated processor controlling the global clock is proposed. To overcome stepping the time at instances where no events are occurring, algorithms to determine for every LP at what point in time the next interaction with another LP shall occur have been developed. Once the minimum timestamp of possible next external events is determined, the global clock can be advanced by $\Delta(S)$, i.e., an amount which depends on the particular state $S$. For a distributed implementation of a global clock [Peac 79], a structured (hierarchical) LP organization can be used [Conc 85] to determine the minimum next event time. A parallel min-reduction operation can bring this timestamp to the root of a process tree [Baik 85], which can then be propagated down the tree. Another possibility is to apply a distributed snapshot algorithm [Chan 85] in order to avoid the bottleneck of a centralized global clock coordinator.

Combinations of synchronous LP simulation with event-driven global clock progression have also been studied. Although the global clock is advanced to the minimum next event time as in the event driven scheme, LPs are only allowed to simulate within a $\Delta$-tick of time, called a bounded lag by Lubachevsky [Luba 88] or a Moving Time Window by [Soko 88].

### 1.5.2 Asynchronous LP Simulation

Asynchronous LP simulation relies on the presence of events occurring at different simulated times that do not affect one another. Concurrent processing of those events thus effectively accelerates sequential simulation execution time.

The critical problem, however, which asynchronous LP simulation poses is the chance of causality errors. Indeed, an asynchronous LP simulation insures correctness if the (total) event ordering as produced by a sequential DES is consistent with the (partial) event ordering as generated by the distributed execution. Jefferson [Jeff 85a] recognized this problem to be the inverse of Lamport’s logical clock problem [Lamp 78], i.e., providing clock values for events occurring in a distributed system such that all events appear ordered in logical time.

It is intuitively convincing and has been shown in [Misr 86] that no causality error can ever occur in an asynchronous LP simulation if and only if every LP adheres to processing events in nondecreasing timestamp order only (local causality constraint (lcc) as formulated in [Fuji 90]). Although sufficient, it is not always necessary to obey the lcc, because two events occurring within one and the same LP may be concurrent (independent of each other) and could thus be processed in any order. The two main categories of mechanisms for asynchronous LP simulation already mentioned adhere to the lcc in different ways: conservative methods strictly avoid lcc violations, even if there is some nonzero probability that an event ordering mismatch will not occur; whereas optimistic methods hazardously use the chance of processing events even if there is nonzero probability for an event ordering mismatch. The variety of mechanisms around these schemes will be the main body of this review.

In a comparison of synchronous and asynchronous LP simulation schemes it has been shown [Feld 90], that the potential performance improvement of an asynchronous LP simulation strategy over the time-stepped variant is at most $O(\log P)$, $P$ being the number of LPs executing concurrently on independent processors. The analysis assumes each time step to take an exponentially distributed amount of execution time $T_{\text{step};i} \sim \exp(\lambda)$ in every LP; $(E[T_{\text{step};i}] = \frac{1}{\lambda})$. As a consequence, the expected simulation time $E[T_{\text{sync}}]$ for a $k$ time step synchronous simulation is $k E[T_{\text{step};i}] = k \frac{1}{\lambda} \sum_{i=1}^{P} \frac{1}{\lambda} \leq k \frac{1}{\lambda} \log(P)$. Relaxing now the synchronization constraint (as an asynchronous sim-
ulation would) the expected simulation time would be $E[T^\text{async}] = E[\max_{i=1..P}(k T_{\text{step},i})] > \frac{k}{\ln P}$.

We have $\lim_{k \to \infty, P \to \infty} \frac{E[T^\text{async}]}{E[T^\text{sync}]} \approx \log(P)$, saying that with increasing simulation size $k$, an asynchronous simulation could complete (at most) $\log(P)$ times as fast as the synchronous simulation, and the maximum attainable speedup of any time stepped simulation is $\frac{E[T^\text{sync}]}{\log(P)}$. These results, however, are a direct consequence of the exponential step execution time assumption, i.e. comparing the expectation of the k-fold sum over the max of exponential random variates (synchronous) with the expectation of the max over P k-stage Erlang random variates. For a step execution time uniformly distributed over $[l, u]$ we have $\lim_{k \to \infty, P \to \infty} \frac{E[T^\text{async}]}{E[T^\text{sync}]} \approx 2$, or intuitively with $T^\text{sync} \leq k u$ and $E[T^\text{async}] \geq k \frac{l+u}{2}$ the ratio of synchronous to asynchronous finishing times is $\frac{E[T^\text{async}]}{E[T^\text{sync}]} \leq 2$, i.e. constant. Therefore for a local event processing time distribution with finite support the improvement of an asynchronous strategy reduces to an amount independent of $P$.

Certainly the model assumptions are far from what would be observed in real implementations on certain platforms, but the results might help to rank the two approaches at least from a statistical viewpoint.

### 2 “Classical” LP Simulation Protocols

#### 2.1 Conservative Logical Processes

LP simulations following a conservative strategy date back to original works by Chandy and Misra [Chand 79] and Bryant [Bryant 84], and are often referred to as the Chandy-Misra-Bryant (CMB) protocols. As described by [Misr 86], in CMB causality of events across LPs is preserved by sending timestamped (external) event messages of type $\langle e \circ t \rangle$, where $ee$ denotes the event and $t$ is a copy of IVT of the sending LP at $(\circ t)$ the instant when the message was created and sent. $t = ts(ee)$ is also called the *timestamp* of the event. A logical process following the conservative protocol (subsequently denoted by LP$^{cons}$) is allowed to process *safe* events only, i.e. events up to a IVT for which the LP has been guaranteed not to receive (external event) messages with IVT $< t$ (timestamp “in the past”). Moreover, all events (internal and external) must be processed in chronological order. This guarantees that the message stream produced by an LP$^{cons}$ is in turn in chronological order, and a communication system (Figure 3) preserving the order of messages sent from LP$^{cons}_i$ to LP$^{cons}_j$ (FIFO) is sufficient to guarantee that no out of chronological order message can ever arrive in any LP$^{cons}$ (necessary for correctness). A conservative LP simulation can thus be seen as a set of all LPs LP$^{cons} = \bigcup_k$ LP$^{cons}_k$ together with a set of directed, reliable, FIFO communication channels CH = $\bigcup_{k,i} (k \neq i) ch_{k,i} = (\text{LP}_k, \text{LP}_i)$ that constitute the *Graph of Logical Processes* GLP$^{cons} = (\text{LP}, \text{CH})$. (It is important to note, that GLP$^{cons}$ has a static topology, which compared to optimistic protocols, prohibits dynamic (re-)scheduling of LPs in a set of physical processors.)

The communication interface CI$^{cons}$ of an LP$^{cons}$ on the input side maintains one input buffer IB$[i]$ and a channel (or link) clock CC$[i]$ for every channel $ch_{i,k} \in \text{CH}$ pointing to LP$^{cons}_k$ (Figure 4). IB$[i]$ intermediates stores arriving messages in FIFO order, whereas CC$[i]$ holds a copy of the timestamp of the message at the head of IB$[i]$; initially CC$[i]$ is set to zero. IVTH = $\min_i$; CC$[i]$ is the time horizon up until which IVT is allowed to progress by simulating internal or external events, since no external event can arrive with a timestamp smaller than IVTH, CI now triggers the SE to conduct event processing just like a (sequential) event driven SE (Figure 1) based on
(internal) events in the EVL, but also to process (external) events from the corresponding IBs respecting chronological order and only up until IVT meets LVTH. During this, SE might have produced future events for remote LPs. For each of those, a message is constructed by adding a copy of IVT to the event, and deposited into FIFO output buffers OB[i] to be picked up there and delivered by the communication system. CI maintains individual output buffers OB[i] for every outgoing channel ch_k,l ∈ CH to subsequent LPs LP_l. The basic algorithm is sketched in Figure 5.

Given now that within the horizon IVTH neither internal nor external events are available to process, then LP_k^{cons} blocks processing, and idles to receive new messages potentially widening the time horizon. Two key problems appear with this policy of “blocking-until-safe-to-process”, namely deadlock and memory overflow as explained with Figure 6: Each LP is waiting for a message to arrive, however, awaiting it from an LP that is blocked itself (deadlock). Moreover, the cyclic waiting of the LPs involved in deadlock leaves events unprocessed in their respective input buffers, the amount of which can grow unpredictably, thus causing memory overflow. This is possible even in the absence of deadlock. Several methods have been proposed to overcome the vulnerability of the CMB protocol to deadlock, falling into the two principle categories: deadlock avoidance and deadlock detection/recory.

2.1.1 Deadlock Avoidance

Deadlock as in Figure 6 can be prevented by modifying the communication protocol based on the sending of nullmessages [Misr 86] of the form <0@0>, where 0 denotes a null event (event without effect). A nullmessage is not related to the simulated model and only serves for synchronization purposes. Essentially it is sent on every output channel as a promise not send any other message with smaller timestamp in the future. It is launched whenever an LP processed an event that did not generate an event message for some corresponding target LP. The receiver LP can use this implicit information to extend its LVTH and by that become unblocked. In our example (Figure 6),
program \( LP_{\text{cons}}(R_k) \)

\[ S_1 \quad \text{IVT} = 0; \quad \text{EVL} = \{\}; \quad S = \text{initialstate}(); \]

\[ S_2 \quad \text{for all } CC[i] \text{ do } (CC[i] = 0) \text{ od}; \]

\[ S_3 \quad \text{for all } ie_i \text{ caused by } S \text{ do } \text{chronological_insert}(\langle ie_i; @\text{occurrence_time}(ie_i) \rangle, \text{EVL}) \text{ od}; \]

\[ S_4 \quad \text{while } \text{IVT} \leq \text{endtime} \text{ do} \]

\[ S_{4.1} \quad \text{for all } IB[i] \text{ do } \text{await not_empty}(IB[i]) \text{ od}; \]

\[ S_{4.2} \quad \text{for all } CC[i] \text{ do } CC[i] = \text{ts(first}(IB[i])) \text{ od}; \]

\[ S_{4.3} \quad \text{IVTH} = \text{min}_{i}CC[i]; \]

\[ S_{4.4} \quad \text{min_channel_index} = i \mid CC[i] = \text{min_channel_clock}; \]

\[ S_{4.5} \quad \text{if } \text{ts(first}(\text{EVL})) \leq \text{IVTH} \]

\[ \quad \begin{align*}
\text{then} & \quad /* \text{select first internal event} */ \\
& \quad e = \text{remove_first}(\text{EVL}); \\
\text{else} & \quad /* \text{select first external event} */ \\
& \quad e = \text{remove_first}(\text{IB[\text{min_channel_index}]})
\end{align*} \]

\[ \text{end if}; \]

\[ S_{4.6} \quad /* \text{now process the selected event} */ \\
\text{IVT} = \text{ts}(e); \]

\[ S_{4.7} \quad \text{if } \text{not nullmessage}(e) \text{ then} \]

\[ S_{4.7.1} \quad S = \text{modified_by_occurrence_of}(e); \]

\[ S_{4.7.2} \quad \text{for all } ie_i \text{ caused by } S \text{ do } \text{chronological_insert}(\langle ie_i; @\text{occurrence_time}(ie_i) \rangle, \text{EVL}) \text{ od}; \]

\[ S_{4.7.3} \quad \text{for all } ie_i \text{ preempted by } S \text{ do } \text{remove}(ie_i, \text{EVL}) \text{ od}; \]

\[ S_{4.7.4} \quad \text{for all } ee_i \text{ caused by } S \text{ do } \text{deposit}(\langle ee_i; @\text{IVT}, \text{corresponding}(\text{OB}[j]) \rangle) \text{ od}; \]

\[ \text{end if}; \]

\[ S_{4.11} \quad \text{for all } \text{empty}(\text{OB}[i]) \text{ do } \text{deposit}(\langle 0; @\text{IVT} + \text{lookahead}(ech_k,i) \rangle, \text{OB}[i]) \text{ od}; \]

\[ S_{4.12} \quad \text{for all } \text{OB}[i] \text{ do } \text{send_out_contents}(\text{OB}[i]) \text{ od}; \]

\[ \text{od while}; \]

Figure 5: Conservative LP Simulation Algorithm Sketch.

---

Communication System

---

\[ \begin{array}{cccc}
29 & 2 & \text{LVT} & 2 \\
19 & 88 & \text{LVT} & 19 \\
14 & 43 & \text{LVT} & 14
\end{array} \]

Figure 6: Deadlock and Memory Overflow

13
after the LP in the middle would have broadcasted \(0@19\) to the neighboring LPs, both of them would have chance to progress their IVT up until time 19, and in turn issue new event messages expanding the IVTHs of other LPs etc. The nullmessage based protocol can be guaranteed to be deadlock free as long as there are no closed cycles of channels, for which a message traversing this cycle cannot increment its timestamp. This implies, that simulation models whose event structure cannot be decomposed into regions such that for every directed channel cycle there is at least one LP to put a nonzero time increment on traversing messages cannot be simulated using CMB with nullmessages.

Although the protocol extension is straight-forward to implement, it can put a dramatic burden of nullmessage overhead on the performance of the LP simulation. Optimizations of the protocol to reduce the frequency and amount of nullmessages, e.g. sending them only on demand (upon request), delayed until some timeout, or only when an LP becomes blocked have been proposed [Misr 86]. An approach where additional information (essentially the routing path as observed during traversal) is attached to the nullmessage, the carrier nullmessage protocol [Cai 90] will be investigated in more detail later.

One problem that still remains with conservative LPs is the determination of when it is safe to process an event. The degree to which LPs can look ahead and predict future events plays a critical role in the safety verification and as a consequence for the performance of conservative LP simulations. In the example in Figure 6, if the LP with IVT 19 could know that processing the next event will certainly increment IVT to 22, then nullmessages \((0@22)\) (so called lookahead of 3) could have been broadcasted as further improvement on the IVTH of the receivers.

Lookahead must come directly from the underlying simulation model and enhances the prediction of future events, which is – as seen – necessary to determine when it is safe to process an event. The ability to exploit lookahead from FCFS queueing network simulations was originally demonstrated by Nicol [Nico 88], the basic idea being that the simulation of a job arriving at a FCFS queue will certainly increment IVT by the service time, which can already be determined, e.g. by random variate presampling, upon arrival since the number of queued jobs is known and preemption is not possible.

### 2.1.2 Example: Conservative LP Simulation of a PN with Model Parallelism

To demonstrate the development and parallel execution of an LP simulation consider again a simulation model described in terms of a PN as depicted in the Figure 7. Assume a physical system consisting of three machines, either being in operation or being maintained. The PN model comprises two places and two transitions with stochastic timing and balanced firing delays \((\tau(T1) \sim exp(0.5), \tau(T2) \sim exp(0.5))\), i.e. time operating is approximately the same as time being maintained. Related to those firing delays and the number of machines being represented by circulating tokens, a certain amount of model parallelism can be exploited when partitioning the net into two LPs, such that the individual PN regions of LP1 and LP2 are: \(R_1 = \{\{T1\}, \{P1\}, \{\{P1, T1\}\}, \tau(T1) \sim exp(\lambda = 0.5)\}\), and \(R_2 = \{\{T2\}, \{P2\}, \{\{P2, T2\}\}, \tau(T2) \sim exp(\lambda = 0.5)\}\).

Let the future list [Nico 88], a sequence of exponentially distributed random firing times (random variates), for T1 and T2 be as in the table of Figure 7. The sequential simulation would then sequence the variates according to their resulting scheduling in virtual time units when simulating the timed behavior of the PN as in Table 1. This sequencing stems from the policy of always using the next free variate from the future list to schedule the occurrence of the next event in EVL. In
an LP simulation scheme this sequencing is related to the protocol applied to maintain causality among the events.

To explain model parallelism as requested by an LP simulation scheme, observe that the firing of a scheduled transition (internal event) always generates an external event, namely a message carrying a token as the event description (token message), and a timestamp equal to the local virtual time IVT of the sending LP. On the other hand, the receipt of an event message (external event) always causes a new internal event to the receiving LP, namely the scheduling of a new transition firing in the local EVL. By just looking at the PN model and the variates sampled in the future list (Figure 7), we observe that the first occurrence of T1 and the first occurrence of T2 could be simulated in a time overlapped way.

This is explained as follows (Figure 8): Both T1 and T2 have infinite server firing semantics, i.e. whenever a token arrives in P1 or P2, T1 (or T2) is enabled with a scheduled firing at IVT plus the transitions next future variate. There are constantly \( M = 3 \) tokens in the PN model, therefore the maximum degree of enabling is \( M \) for both T1 and T2. Considering now the initial state \( S = (2, 1) \) (two tokens in P1 and one in P2), one occurrence of T1 is scheduled for time \( t_{T1} = 0.17 \), and another one for \( t'_{T1} = 0.37 \). One occurrence of T2 is scheduled for time \( t_{T2} = 0.51 \). The next variate for T1 is 0.22, the one for T2 is 0.39. A token can be expected in P1 at \( \min(0.51, 0.39, 0.42) = 0.39 \) at the earliest, leading to a new (the third) scheduling of T1 at \( 0.39 + 0.22 = 0.61 \) at the earliest, maybe later. Consequently the first occurrence of T1 must be at \( t(T1_1) = 0.17 \), and the second occurrence of T1 must be \( t(T1_2) = 0.37 \). The first occurrence of T2 can be either the one scheduled

![Figure 7: LP Simulation of a Trivial PN with Model Parallelism](image_url)
Since \( T_1 \) interfere with each other and can therefore be simulated independently, the first occurrence of \( T_1 \) at 0.17 + 0.39 = 0.56, or the one invoked by the second occurrence of \( T_1 \) at 0.37 + 0.42 = 0.78. Clearly, the first occurrence of \( T_2 \) must be at \( t(T_2) = 0.17 + 0.39 = 0.56 \), and the second occurrence of \( T_2 \) must be at \( t(T_2) = 0.17 + 0.39 = 0.56 \), etc. Since \( T_1 \sim T_2 \) with \( t(T_2) < t(T_2) \) and \( T_2 \sim T_1 \) with \( t(T_1) < t(T_1) \), \( T_1 \) and \( T_2 \) do not interfere with each other and can therefore be simulated independently (\( T_1 \rightarrow T_2 \) denotes the \textit{direct scheduler} causality of the \( i \)-th occurrence of \( T_1 \) onto the \( j \)-th occurrence of \( T_2 \)).

As was seen, the model that we consider in Figure 7 provides inherent model parallelism. In order to exploit this model parallelism in a CMB simulation, the PN model is decomposed into two regions \( R_1 \) and \( R_2 \), which are assigned to two LPs \( LP_1 \) and \( LP_2 \), such that \( GLP = (\{LP_1, LP_2\}, \{ch_{1,2}, ch_{2,1}\}) \), where the channels \( ch_{1,2} \) and \( ch_{2,1} \) are supposed to substitute the PN arcs (\( T_1, P_2 \)) and (\( T_2, P_1 \)) respectively. Both \( ch_{1,2} \) and \( ch_{2,1} \) carry messages containing tokens that were generated by the firing of a transition in a remote LP. Consequently, \( ch_{1,2} \) propagates a message of the form \( m = \langle 1, P_2, t \rangle \) from \( LP_1 \) to \( LP_2 \) on the occurrence of a firing of \( T_1 \), in order to deposit 1 (first component of \( m \)) token into place \( P_2 \) (second component of \( m \)) at time \( t \) (third component). The timestamp \( t \) is produced as a copy of the IVT of \( LP_1 \) at the instant of that firing of \( T_1 \), that produced the token.

A CMB \textit{parallel} execution of the LP simulation model developed above, since operating in a synchronous way in two phases (first simulate one event locally, then transmit messages), generates the trace in Table 2. In step 0, both LPs use precomputed random variates from their individual future lists and schedule events. In step 1, no event processing can happen due to IVTH = 0.0, LPs are blocked (see indication in B column). Generally in such a situation every LP \( i \) computes its lookahead \( la(ch_{i,j}) \) imposed on the individual output channels \( j \). In the example we have

\[
la(ch_{i,j}) = \min\( (IVT_i - \min_{k=1..M} s_j(st_k)) \) \quad \text{min}_{k=1..(M-S)} flk
\]

where \( st_k \) is the scheduled occurrence time of the \( k \)-th entry in EVL, \( fl_k \) is the \( k \)-th free variate in the future list, and \( M \) is the maximum enabling degree (tokens in the PN model). For example, the lookahead in \( LP_1 \) in the state of step 1 imposed on the channel to \( LP_2 \) is \( la(ch_{1,2}) = 0.17 \), whereas \( la(ch_{2,1}) = 0.39 \). \( la \) is now attached to the LP’s IVT, giving the timestamps for the nullmessage \( \langle 0; P_2; 0.17 \rangle \) sent from \( LP_1 \) to \( LP_2 \), and \( \langle 0; P_1; 0.39 \rangle \) sent from \( LP_2 \) to \( LP_1 \). The latter, when arriving at \( LP_1 \), unblocks the SE, such that the first event out of EVL can be processed, generating the
event message \((1; P; 0.17)\). This message, however, as received by \(LP_2\) still cannot unblock \(LP_2\) since it carries the same timestamp as the previous nullmessage; also the local lookahead cannot be improved and \((0; P; 0.51)\) is resent. It takes another iteration to finally unblock \(LP_2\), which can then process its first event in step 5, etc. It is easy seen from the example, that the CMB protocol (for the particular example) forces a ‘logical’ barrier synchronization whenever the sequential DES (see trace in Table 1) switches from processing a T1 related event to a T2 related one and vice versa (at VT 0.17, 0.51, 0.73, etc.). In the diagram in Figure 8, this is at points where the arrow denoting a token move from T1 (T2) to T2 (T1) has the opposite direction that the previous one.

### 2.1.3 Deadlock Detection/Recovery

An alternative to the Chandy-Misra-Bryant protocol avoiding nullmessages has also been proposed by Chandy and Misra [Chan 81], allowing deadlocks to occur, but providing a mechanism to detect it and recover from it. Their algorithm runs in two phases: (i) *parallel phase*, in which the simulation runs until it deadlocks, and (ii) *phase interface*, which initiates a computation allowing some LP to advance IVT. They prove, that in every parallel phase at least one event will be processed generating at least one event message, which will also be propagated before the next deadlock. A central controller is assumed in their algorithm, thus violating a distributed computing principle. To avoid a single resource (controller) to become a communication performance bottleneck during deadlock detection, any general distributed termination detection algorithm [Matt 87] or distributed deadlock detection algorithm [Chan 83] could be used instead.

In an algorithm described by Misra [Misr 86], a special message called *marker* circulates through GILP to detect and correct deadlock. A cyclic path for traversing all \(ch_{i,j} \in CH\) is precomputed and LPs are initially colored *white*. An LP that received the marker takes the color *white* and is supposed to route it along the cycle in *finite* time. Once an LP has either received or sent an event message since passing the marker, it turns to *red*. The marker identifies deadlock if the last \(N\) LPs visited were all *white*. Deadlock is properly detected as long as for any \(ch_{i,j} \in CH\) all messages sent over \(ch_{i,j}\) arrive at \(LP_j\) in the time order as sent by \(LP_i\). If the marker also carries the next event times of visited *white* LPs, it knows upon detection of deadlock the smallest next event time as well as the LP in which this event is supposed to occur. To recover from deadlock, this LP is...
invoked to process its first event. Obviously message lengths in this algorithm grow proportionally to the number of nodes in GLP.

Bain and Scott [Bain 88] propose an algorithm for demand driven deadlock free synchronization in conservative LP simulation that avoids message lengths to grow with the size of GLP. If an LP wants to process an event with timestamp \( t \), but is prohibited to do so because \( \text{CC}[j] < t \) for some \( j \), then it sends *time requests* containing the sender's process id and the requested time \( t \) to all predecessor LPs with this property. The predecessors, however, may have already advanced their IVT in the mean time. Predecessors are supposed to inform the sender LP when they can guarantee that they will not emit an event message at a time lower than the requested time \( t \). Three types of reply types are used to avoid repeated polling in the presence of cycles: a *yes* indicates that the predecessor has reached the requested time, a *no* indicates that it has not (in which case another request must be made), and a *ryes* ("reflected yes") indicates that it has conditionally reached \( t \). *Ryes* replies, together with a *request queue* maintained in every LP, essentially have the purpose to detect cycles and to minimize the number of subsequent requests sent to predecessors. If the process id and time of a request received match any request already in the request queue, a cycle is detected and *ryes* is replied. Otherwise, if the LP's IVT equals or exceeds the requested time a *yes* is replied, whereas if the LP's IVT is less the requested time the request is enqueued in the request queue, and request copies are recursively sent to the receiver's predecessors with \( \text{CC}[i]'s < t \), etc. The request is complete when all channels have responded, and the request reached the head of the request queue. At this time the request is removed from the request queue and a reply is sent to the requesting LP. The reply to the successor from which the request was answered is *no* (*ryes*), if any request to a predecessor was answered with *no* (*ryes*), otherwise *yes* is sent. If *no* was received in an LP initiating a request, the LP has to restart the time request with lower channel clocks.

The *time-of-next-event algorithm* as proposed by Groselj and Tropper [Gros 88] assumes more than one LP mapped onto a single physical processor, and computes the greatest lower bound of the timestamps of the event messages expected to arrive next at all empty links on the LPs located at that processor. It thus helps to unblock LPs within one processor, but does not prevent deadlocks across processors. The lower bound algorithm is an instance of the single source shortest path problem.

### 2.1.4 Conservative Time Windows

Conservative LP simulations as presented above are distributed in nature since LPs can operate in a totally asynchronous way. One way to make these algorithms more synchronous in order to gain from the availability of fast synchronization hardware in multiprocessors is to introduce a *window* \( W_i \) in simulated time for each LP \( i \), such that events within this *time window are safe* (events in \( W_i \) are independent of events in \( W_j, i \neq j \)) and can be processed concurrently across all LP; [Luba 88], [Nico 91].

A conservative time window (CTW) parallel LP simulation synchronously operates in two phases. In phase (i) (*window identification*) for every LP \( i \), a chronological set of events \( W_i \) is identified such that for every event \( e \in W_i \), \( e \) is causally independent of any \( e' \in W_j, j \neq i \). Phase (i) is accomplished by a barrier synchronization over all LPs. In phase (ii) (*event processing*) every LP \( i \) processes events \( e \in W_i \) sequentially in chronological order. Again, phase (ii) is accomplished by a barrier synchronization. Since the algorithm iteratively lock-steps over the two consecutive
phases, the hope to gain speedup over a purely sequential DES heavily depends on the efficiency of the synchronization operation on the target architecture, but also on the event structure in the simulation model. Different windows will generally have different cardinality of the covered event set, maybe some windows will remain empty after the identification phase for one cycle. In this case the corresponding LPs would idle for that cycle.

A considerable overhead can be imposed on the algorithm by the identification of when it is safe to process an event within LP; (window identification phase). Lubachevsky [Luba 88] proposes to reduce the complexity of this operation by restricting the lag on the LP simulation, i.e. the difference in occurrence time of events being processed concurrently is bounded from above by a known finite constant (bounded lag protocol). By this restriction, and assuming a "reasonable" amount of dispersion of events in space and time, the execution of the algorithm on N processors in parallel will have one event processed in $O(\log N)$ time on average. An idealized message passing architecture with a tree-structured synchronization network supporting an efficient realization of the bounded lag restriction is assumed for the analysis.

2.1.5 The Carrier Null Message Protocol

Another approach to reduce the overwhelming amount of null messages occurring with the CMB protocol is to add more information to the null messages. The carrier null message protocol [Cai 90] uses null messages to advance CC[i]'s and acquire/propagate knowledge global to the participating LPs, with the goal of improving the ability of lookahead to reduce the message traffic.

Indeed, good lookahead can reduce the number null messages as is motivated by the example in Figure 9, where a source process produces objects in constant time intervals $\omega = 50$. The join, pass and split processes manipulate objects, consuming 2 time units per object. Eventually objects are released from split into sink. For the example we have $la(ch_{ij}) = 2 \forall i, j \in \{source, join, pass, split, sink\}$, $(i \neq j)$, except $la(ch_{source,join}) = 50$. After the first object release into LP;join, all LPs except LPsource are blocked, and therefore start propagating local lookahead via null messages. After the propagation of (overall) 4 nullmessages all LPs beyond LPsource have progressed IVT's and CC's to 2. It shall take further 96 nullmessages until LP;join can make its first object manipulation, and after that another 100 for the second object, etc. If LP;join could have learned that it had just waited for itself, it could have immediately simulated the external event (with VT 50). Besides the importance of the availability of global information within the LPs, the impact of lookahead onto LP simulation performance is now also easily seen: the smaller the lookahead in the successor LPs, the higher the communication overhead caused by nullmessages, the higher also the performance degrade.

To generally realize such a waiting dependency across LPs the CNM protocol employs additional nullmessages of type $\langle c0, t, R, la.inf \rangle$, where $c0$ is an identification as a carrier nullmessage, $t$ is the timestamp, $R$ is information about the travelling route of the message and $la.inf$ is lookahead information. Once LP;join had received a carrier nullmessage with its id as source and sink in $R$, it can be sure (but only in the particular example) not to receive an event message via that path, unless LP;join itself had sent an event message along that path. So it can - without further waiting - after having received the first carrier nullmessage process the event message from LPsource, and thus increment the CC's and IVT's of all successors on the route in $R$ considerably.

Should there be any other "source"-like LP entering event messages into the waiting dependency loop, the arguments above are no longer valid. For this case it is in fact not sufficient to only carry
the route information with the nullmessage, but also the earliest time of possible event messages that would break the cyclic waiting dependency. Exactly this information is carried by \( la.inf \), the last component in the carrier nullmessage.

### 2.2 Optimistic Logical Processes

Optimistic LP simulation strategies, in contrast to conservative ones, do not strictly adhere to the local causality constraint \( lcc \) (see Section 1.5.2), but allow the occurrence of causality errors and provide a mechanism to recover from \( lcc \) violations. In order to avoid blocking and safe-to-process determination which are serious performance pitfalls in the conservative approach, an optimistic LP progresses simulation (and by that advances IVT) as far into the simulated future as possible, without warranty that the set of generated (internal and external) events is consistent with \( lcc \), and regardless to the possibility of the arrival of an external event with a timestamp in the local past.

#### 2.2.1 Time Warp

Pioneering work in optimistic LP simulation was done by Jefferson and Sowizral [Jeff 85b, Jeff 85a] in the definition of the Time Warp (TW) mechanism, which like the Chandy-Misra-Bryant protocol uses the sending of messages for synchronization. Time Warp employs a rollback (in time) mechanism to take care of proper synchronization with respect to \( lcc \). If an external event arrives with timestamp in the local past, i.e. out of chronological order (straggler message), then the Time Warp scheme rolls back to the most recently saved state in the simulation history consistent with the timestamp of the arriving external event, and restarts simulation from that state on as a matter of \( lcc \) violation correction. Rollback, however, requires a record of the LP’s history with respect to the simulation of internal and external events. Hence, an LP\(^{opt} \) has to keep sufficient internal state information, say a state stack SS, which allows for restoring a past state. Furthermore, it has to administrate an input queue IQ and an output queue OQ for storing messages received and sent. For reasons to be seen, this logging of the LP’s communication history must be done in chronological order.
order. Since the arrival of event messages in increasing time stamp order cannot be guaranteed, two different kinds of messages are necessary to implement the synchronization protocol: first the usual external event messages \( m^+ = (ee @ t, +) \), (where again \( ee \) is the external event and \( t \) is a copy of the senders LVT at the sending instant) which will subsequently call positive messages. Opposed to that are messages of type \( m^- = (ee @ t, -) \) called negative or antimessages, which are transmitted among LPs as a request to annihilate the prematurely sent positive message containing \( ee \), but for which it meanwhile turned out that it was computed based on a causally erroneous state.

The basic architecture of an optimistic LP employing the Time Warp rollback mechanism is outlined in Figure 10. External events are brought to some LP\(_k\) by the communication system in much the same way as in the conservative protocol. Messages, however, are not required to arrive in the sending order (FIFO) in the optimistic protocol, which weakens the hardware requirements for executing Time Warp. Moreover, the separation of arrival streams is also not necessary, and so there is only a single IB and a single OB (assuming that the routing path can be deduced from the message itself). The communication related history of LP\(_k\) is kept in IQ and OQ, whereas the state related history is maintained in the SS data structure. All those together represent CI\(_k\); SE\(_k\) is an event driven simulation engine equivalent to the one in LP\(_{cons}\).

The triggering of CI to SE is sketched with the basic algorithm for LP\(_{opt}\) in Figure 11. The LP mainly loops (S3) over four parts: (i) an input-synchronization to other LP\(_s\) (S3.1), (ii) local event processing (S3.2 – S3.8), (iii) the propagation of external effects (S3.9) and (iv) the (global) confirmation of locally simulated events (S3.10 – S3.11). Part (ii) and (iii) are almost the same

Figure 10: Architecture of an Optimistic Logical Process
program $LP_{opt}(R_i)$

$S1$  
GVT = 0; IVT = 0; EVL = \{\}; $S$ = initialstate();

$S2$  
for all $i \epsilon_i$ caused by $S$ do chronological_insert($(i \epsilon_i; occurrence\_time(i \epsilon_i)),$ EVL) od;

$S3$  
while GVT < endtime do

$S3.1$  
for all $m \epsilon IB$ do

$S3.1.1$  
if $ts(m) < IVT$ /* $m$ potentially affects local past */
then
if (positive($m$) and dual($m$) $\notin$ IQ) or (negative($m$) and dual($m$) $\in$ IQ) /* rollback */
restore_earliest_state before($ts(m)$);
geninate_and_sendout (antimesages);
IVT = earliest_state_timestamp before($m$);
endif;
endif; /* irrespective of how $m$ is related to IVT */

$S3.1.2$  
if dual($m$) $\in$ IQ
then remove(dual($m$), IQ); /* annihilate */
else chronological_insert(external_event($m$)$@ts(m)$, $sign(m)$), IQ);
endif;

od;

$S3.2$  
if $ts$((first(EVL)) $\leq$ ts((first_nonnegative(IQ)))
then $e$ = remove_first(EVL); /* select first internal event*/
else $e$ = remove_first_nonnegative(IQ); /* select first external event*/
endif;
/* now process the selected event */

$S3.3$  
IVT = $ts(e)$;

$S3.4$  
$S$ = modified_by_occurrence_of($e$);

$S3.5$  
for all $i \epsilon_i$ caused by $S$ do chronological_insert($(i \epsilon_i; occurrence\_time(i \epsilon_i))$, EVL) od;

$S3.6$  
for all $i \epsilon_i$ preempted by $S$ do remove($i \epsilon_i$, EVL) od;

$S3.7$  
log_new_state($(IVT$, $S$, copy_of(EVL)), SS);

$S3.8$  
for all $e \epsilon_i$ caused by $S$ do
  deposit($(e \epsilon_i; @IVT, +))$, OB);
  chronological_insert($(e \epsilon_i; @IVT, +)$, OQ);
  od;

$S3.9$  
send_out_contents(OB);

$S3.10$  
GVT = advance_GVT();

$S3.11$  
fossil_collection(GVT);

od while;

Figure 11: Optimistic LP Simulation Algorithm Sketch.
Arriving Message is of Type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m^+$</td>
<td>$dual m^- \text{ exists in IQ}$</td>
</tr>
<tr>
<td></td>
<td>annihilate $dual m^-$</td>
</tr>
<tr>
<td></td>
<td>chronological insert ($m^+$, IQ)</td>
</tr>
<tr>
<td></td>
<td>$dual m^-$ does NOT exist</td>
</tr>
<tr>
<td>$m^-$</td>
<td>$dual m^+ \text{ exists in IQ (not yet processed)}$</td>
</tr>
<tr>
<td></td>
<td>annihilate $dual m^+$</td>
</tr>
<tr>
<td></td>
<td>chronological insert ($m^-$, IQ)</td>
</tr>
<tr>
<td></td>
<td>$dual m^-$ does NOT exist</td>
</tr>
<tr>
<td></td>
<td>$dual m^+ \text{ exists in IQ (already processed)}$</td>
</tr>
<tr>
<td></td>
<td>rollback</td>
</tr>
<tr>
<td></td>
<td>annihilate $dual m^+$</td>
</tr>
<tr>
<td></td>
<td>chronological insert ($m^+$, IQ)</td>
</tr>
<tr>
<td></td>
<td>$dual m^- \text{ does NOT exist}$</td>
</tr>
</tbody>
</table>

timestamp(m) $\geq$ LVT (in the local future)

timestamp(m) $<$ LVT (in the local past)

Figure 12: The Time Warp Message based Synchronization Mechanism

as was seen with $\text{LP}^{cons}$. The input synchronization ($\text{Rollback and Annihilation}$) and confirmation ($\text{GVT}$) part however are the key mechanisms in optimistic LP simulation.

2.2.2 Rollback and Annihilation Mechanisms

The input synchronization of $\text{LP}^{opt}$ (rollback mechanism) relates arriving messages to the current value of the LP’s IVT and reacts accordingly (see Figure 12). A message affecting the LP’s “local future” is moved from the IB to the IQ respecting the timestamp order, and the encoded external event will be processed as soon as IVT advances to that time. $\langle ee3@7, + \rangle$ in the IB in Figure 10 is an example of such an unproblematic message (IVT = 6). A message timestamped in the “local past” however is an indicator of a causality violation due to tentative event processing. The rollback mechanism ($S3.1.1$) in this case restores the most recent $\text{loc}$-consistent state, by reconstructing $S$ and EVL in the simulation engine from copies attached to the SS in the communication interface. Also IVT is warped back to the timestamp of the straggler message. This so far has compensated the local effects of the $\text{loc}$ violation; the external effects are annihilated by sending an antimessage for all previously sent outputmessages (in the example $\langle ee6@6, - \rangle$ and $\langle ee1@7, - \rangle$ are generated and sent out, while at the same time $\langle ee6@6, + \rangle$ and $\langle ee1@7, + \rangle$ are removed from OQ). Finally, if a negative message is received (e.g. $\langle ee2@6, - \rangle$) it is used to annihilate the dual positive message ($\langle ee2@6, + \rangle$) in the local IQ. Two cases for the negative messages must be distinguished: (i) If the dual positive message is present in the receiver IQ, then this entry is deleted as an annihilation. This can be easily done if the positive message has not yet been processed, but requires rollback if it was. (ii), if a dual positive message is not present (this case can only arise if the communication
system does not deliver messages in a FIFO fashion), then the negative message is inserted in IQ (irrespective of its relation to IVT) to be annihilated later by the (delayed) positive message still in traffic.

As is seen now, the rollback mechanism requires a periodic saving of the states of SE (IVT, S and EVL) in order to able to restore a past state (S3.7), and to log output messages in OQ to be able to undo propagated external events (S3.8). Since antimeessages can also cause rollback, there is the chance of rollback chains, even recursive rollback if the cascade unrolls sufficiently deep on a directed cycle of GLP. The protocol however guarantees, although consuming considerable memory and communication resources, that any rollback chain eventually terminates whatever its length or recursive depth is.

**Lazy Cancellation**  In the original Time Warp protocol as described above, an LP receiving a straggler message initiates sending antimeessages immediately when executing the rollback procedure. This behavior is called aggressive cancellation. As a performance improvement over aggressive cancellation, the lazy cancellation policy does not send an antimeessage \( m^- \) for \( m^+ \) immediately upon receipt of a straggler. Instead, it delays its propagation until the resimulation after rollback has progressed to IVT = \( ts(m^+) \) producing \( m^+ = m^- \). If the resimulation produced \( m^+ = m^- \), no antimeessage has to be sent all [Gafn 88a]. Lazy cancellation thus avoids unnecessary cancelling of correct messages, but has the liability of additional memory and bookkeeping overhead (potential antimeessages must be maintained in a rollback queue) and delaying the annihilation of actually wrong simulations.

Lazy cancellation can also be based on the utilization of lookahead available in the simulation model. If a straggler \( m^+ < \) IVT is received, then obviously antimeessages do not have to be sent for messages \( m \) with timestamp, \( ts(m^+) \leq ts(m) < ts(m^+) + la \). Moreover, if \( ts(m^+) + la \geq \) IVT even rollback does not need to be invoked. As opposed to lookahead computation in the CMB protocol, lazy cancellation can exploit implicit lookahead, i.e., does not require its explicit computation.

The traces in Figure 3 represent the behavior of the optimistic protocol with the lazy cancellation message annihilation in a parallel LP simulation of the PN model in Figure 7. (The trace table is to be read in the same way as the one in Figure 2, except that there is a rollback indicator column RB instead of a blocking column B.) In step 2, for example, LP \(_2\) receives the straggler \((1; \text{P}1; 0.17)\) at IVT = 0.51. Message annihilation and rollback can be avoided due to the exploitation of the lookahead from the next random variate in the future list, 0.39. The effect of the straggler is in the future of LP \(_2\) (0.56).

It has been shown [Jeff 91] that Time Warp with lazy cancellation can produce so called “supercritical speedup”, i.e., surpass the simulations critical path by the chance of having wrong compu-
tations produce correct results. By immediately discarding rolled back computations this chance is lost for the aggressive cancellation policy. A performance comparison of the two, however, is related to the simulation model. Analysis by Reiher and Fujimoto [Reih 90] shows that lazy cancellation can arbitrarily outperform aggressive cancellation and vice versa, i.e., one can construct extreme cases for lazy and aggressive cancellation such that if one protocol executes in $\alpha$ time using $N$ processors, the other uses $\alpha N$ time. Nevertheless, empirical evidence is reported “slightly” in favor of lazy cancellation for certain simulation applications.

Lazy Reevaluation Much like lazy cancellation delays the annihilation of external effects upon receiving a straggler at IVT, lazy re-evaluation delays discarding entries on the state stack SS. Should the recomputation after rollback to time $t <$ IVT reach a state that exactly matches one logged in SS and the IQ is the same as the one at that state, then immediately jump forward to IVT, the time before rollback occurred. Thus, lazy reevaluation prevents from the unnecessary recomputation of correct states and is therefore promising in simulation models where events do not modify states (“read-only” events). A serious liability of this optimization is again additional memory and bookkeeping overhead, but also (and mainly) the considerable complication of the Time Warp code [Fuj 90]. To verify equivalence of IQ’s the protocol must draw and log copies of the IQ in every state saving step ($SS_3$). In a weaker lazy re-evaluation strategy one could allow jumping forward only if no message has arrived since rollback.

Lazy Rollback The difference of virtual time in between the straggler $m^*$, $ts(m^*)$, and its actual effect at time $ts(m^*) + la(\epsilon e)$ $\geq$ IVT can again be overjumped, saving the computation time for the resimulation of events in between $[ts(m^*), ts(m^*) + la(\epsilon e)]$. $la(\epsilon e)$ is the lookahead imposed by the external event carried by $m^*$.

Breaking/Preventing Rollback Chains Besides the postponing of erroneous message and state annihilation until it turns out that they are not reproduced in the repeated simulation, other techniques have been studied to break cascades of rollbacks as early as possible. Prakash and Subramanian [Prak 91], comparable to the carrier null message approach, attach a limited amount of state information to messages to prevent recursive rollbacks in cyclic GLPs. This information allows LPs to filter out messages based on preempted (obsolete) states to be eventually annihilated by chasing antimessages currently in transit. Related to the (conservative) bounded lag algorithm, Lubachevsky, Shwartz and Weiss have developed a filtered rollback protocol [Luba 91] that allows optimistically crossing the lag bound, but only up to a time window upper edge. Causality violations can only affect the time period in between the window edge and the lag bound, thus limiting (the relative) length of rollback chains. The SRADS protocol by Dickens and Reynolds [Dick 90], although allowing optimistic simulation progression, prohibits the propagation of uncommitted events to other LPs. Therefore, rollback can only be local to some LP and cascades of rollback can never occur. Madisetti, Walrand and Messerschmitt with their protocol called Wolf-calls freeze the spatial spreading of uncommitted events in so called spheres of influence $W(LP_1, \tau)$, defined as the set of LPs that can be influenced by a message from LP$_1$ at time $ts(m) + \tau$ respecting computation times $a$ and communication times $b$. The Wolf algorithm ensures that the effects of an uncommitted event generated by LP$_1$ are limited to a sphere of a computable (or selectable) radius around LP$_1$, and the number of broadcasts necessary for a complete annihilation within the
sphere is bounded by a computable (or chooseable) number of steps \( B \) \( (B \text{ being provably smaller than for the standard Time Warp protocol}) \).

### 2.2.3 Optimistic Time Windows

A similar idea of "limiting the optimism" to overcome rollback overhead potentials is to advance computations by "windows" moving over simulated time. In the original work of Sokol, Briscoe and Wieland [Soko 88], the moving time window (MTW) protocol, neither internal nor external events \( e \) with \( ts(e) > t + \Delta \) are allowed to be simulated in the time window \([t, t + \Delta] \), but are postponed for the next time window \([t + \Delta, t + 2\Delta] \). Two events \( e \) and \( e' \) timestamped \( ts(e) \) and \( ts(e') \) respectively therefore can only be simulated in parallel iff \( |ts(e) - ts(e')| < \Delta \). Naturally, the protocol is in favor of simulation models with a low variation of event occurrence distances relative to the window size. Compared to a time-stepped simulation, MTW does not await the completion of all events \( e \) with \( t \leq ts(e) < t + \Delta \) which would cause idle processors at the end of each time window, but invokes an attempt to move the window as soon as the number of events to be executed falls below a certain threshold. In order to keep moving the time window, LPs are polled for the timestamp of their earliest next event \( t_i(e) \) (polling takes place simultaneously with event processing) and the window is advanced to \( \min(t_i(e), \min(t_i(e) + \Delta) \). (The next section will show the equivalence of the window lower edge determination to GVT computation.) Obviously the advantage of MTW and related protocols is the potential effective implementation as a parallel LP simulation, either on a SIMD architecture or in a MIMD environment where the reduction operation \( \min(t_i(e) \) can be computed utilizing synchronization hardware. Points of criticism are the assumption of approximately uniform distribution of event occurrence times in space and the ignorance with respect to potentially "good" optimism beyond the upper window edge. Furthermore, a natural difficulty is the determination of the \( \Delta \) admitting enough events to make the simulation efficient.

The latter is addressed with the adaptive Time Warp concurrency control algorithm (ATW) proposed by Ball and Hyyt [Ball 90], allowing the window size \( \Delta(t) \) be adapted at any point \( t \) in simulation time. ATW aims to temporarily suspend event processing if it has observed a certain amount of \( lec \) violations in the past. In this case the LP would conclude that it progresses IVT too fast compared to the predecessor LPs and would therefore stop IVT advancement for a time period called the blocking window (BW). BW is determined based on the minimum of a function describing wasted computation in terms of time spent in a (conservatively) blocked mode, or a fault recovery mode as induced by the Time Warp rollback mechanism.

### 2.2.4 The Limited Memory Dilemma

All arguments on the execution of the Time Warp protocol so far assumed the availability of a sufficient amount of free memory to record internal and external effect history for pending rollbacks, and all arguments were related to the time complexity. Indeed, Time Warp with certain memory management strategies to be described in the sequel can be proven to work correctly when executed with \( O(M_{seq}) \) memory, where \( M_{seq} \) is the number of memory locations utilized by the corresponding sequential DES. Opposed to that, the CMB protocol may require \( O(kM_{seq}) \) space, but may also use less storage than sequential simulation, depending on the simulation model (it can even be proven that simulation models exist such that the space complexity of CMB is \( O((M_{seq})^k) \)). Time Warp always consumes more memory than sequential simulation [Lin 91], and a memory limitation imposes a performance decrease on Time Warp: providing just the minimum of memory necessary
may cause the protocol to execute fairly slow, such that the memory/performance tradeoff becomes an issue.

Memory management in Time Warp follows two goals: (i) to make the protocol operable on real multiprocessors with bounded memory, and (ii) to make the execution of Time Warp performance efficient by providing “sufficient” memory. An infrequent or incremental saving of history information in some cases can prevent, maybe more effectively than one of the techniques presented for limiting the optimism in Time Warp, aggressive memory consumption. Once, despite the application of those techniques, available memory is exhausted, fossil collection could be applied as a technique to recover memory used for history recording that will definitely not be used anymore due to an assured lower bound on the timestamp of any possible future rollback (GVT). Finally, if even fossil collection fails to recover enough memory to proceed with the protocol, additional memory could be freed by returning messages from the IQ (message sendback, cancelback) or invoking an artificial rollback reducing space used for storing the OQ.

2.2.5 Incremental and Interleaved State Saving

Minimizing the storage space required for simulation models with complex sets of state variables \( S_i \subset S \), \( S_i \) being the subset stored and maintained by LP; that do not extensively change values over IVT progression, can be accomplished effectively by just saving the variables \( s_j \in S_i \) affected by a state change. This is mainly an implementation optimization upon step S3.4 in the algorithm in Figure 11. This incremental state saving can also improve the execution complexity in step S3.7, since generally less data has to be copied into the logrecord. Obviously the same strategy could be followed for the EVL, or the IQ in a lazy reevaluation protocol. Alternatively, imposing a condition upon step S3.7:

\[
S3.7 \quad \text{if } \text{(step\_count \ modulo \ } \pi) == 0 \text{ then } \text{log\_new\_state((IVT, S, copy\_of\_EVL)), SS);}
\]

could be used to interleave the continuity of saved states and thus on the average reduce the storage requirement to \( \frac{1}{2} \) of the noninterleaved case.

Both optimizations, however, increase the execution complexity of rollback. In incremental state
saving protocols, desired states have to be reconstructed from increments following back a path further into the simulated past than required by rollback itself. The same is true for interleaved state saving, where the most recent saved state older than the straggler must be searched for, a reexecution up until the timestamp of the straggler (coast forward) must be started which is a clear waste of CPU cycles since it just reproduces states that have already been computed but were not saved, and finally the straggler integration and usual reexecution are necessary (Figure 13). The tradeoff between state saving costs and the coast forward overhead has been studied (as reported by [Nico 94 in reference to Lin and Lazowska] based on expected event processing time \( \epsilon = E[exec(e)] \) and state saving costs \( \sigma \), giving an optimal interleaving factor \( \pi^* \) as

\[
\left\lfloor \sqrt{(\alpha - 1)\beta} \right\rfloor < \pi^* < \left\lfloor \sqrt{(2\alpha + 1)\beta} \right\rfloor
\]

where \( \alpha \) is the average number of rollbacks with \( \pi = 1 \) and \( \beta = \pi^* \). The result expresses that an overestimation of \( \pi^* \) is more severe to performance than an underestimation by the same (absolute) amount. In a study of the optimal checkpointing interval explicitly considering state saving and restoration costs while assuming \( \pi \) does neither affect the number of rollbacks nor the number of rolled back events in [Lin 93], an algorithm is developed that, integrated into the protocol, “on-the-fly”, within a few iterations, automatically adjusts \( \pi \) to \( \pi^* \). It has been shown that some \( \pi \), though increasing the rollback overhead, can reduce overall execution time.

2.2.6 Fossil Collection

Opposed to techniques that reclaim memory temporarily used for storing events and messages related to the future of some LP, fossil collection aims to return space used by history records that will no longer be used by the rollback synchronization mechanism. To assure from which state in the history (and back) computations can be considered fully committed, the determination of the value of global virtual time (GVT) is necessary.

Consider the tuple

\[ \Sigma_i(T) = (IVT_i(T), IQ_i(T), SS_i(T), OQ_i(T)) \]

to be a local snapshot of LP_i at real time T, i.e. IVT_i, IQ_i is the input queue as seen by an external observer at real time T, etc., and \( \Sigma(T) = \bigcup_{i=1}^{N} \Sigma_i(T) \cup CS(T) \) be the global snapshot of GLP. Further let IVT_i(T) be the local virtual time in LP_i, i.e. the timestamp of the event being processed at the observation instant T, and \( UM_{i,j}(T) \) the set of external events imposed by LP_i upon LP_j encoded as messages m in the snapshot \( \Sigma \). This means m is either in transit on channel ch_i,j in CS or stored in some IQ_j, but not yet processed at time T. Then the GVT at real time T is defined to be:

\[ GVT(T) = \min(\min_{i} IVT_i(T), \min_{i,j,m} ts(m)) \]

It should be clear even by intuition, that at any real time T, GVT(T) represents the maximum lower bound to which any rollback could ever backdate IVT_i (\( \forall i \)). An obvious consequence is that any processed event e with \( ts(e) < GVT(T) \) can never (at no instant T) be rolled back, and can therefore be considered as (irrevocably) committed [Lin 91] (Figure 13). Further consequences (for all LP_i) are that:
(i) messages \( m \in \text{IQ} \) with \( ts(m) \leq \text{GVT}(T) \), as well as messages \( m \in \text{OQ} \) with \( ts(m) \leq \text{GVT}(T) \) are obsolete and can be discarded (from IQ, OQ) after real time \( T \).

(ii) state variables \( s \in S_i \) stored in \( \text{SS}_i \) as with \( ts(s) \leq \text{GVT}(T) \) are obsolete and can be discarded after real time \( T \).

Making use of these possibilities, i.e. getting rid of external event history according to (i) and of internal event history according to (ii) that is no longer needed to reclaim memory space is the idea behind fossil collection. It is called as a procedure in the abstracted Time Warp algorithm (Figure 11) in step S3.11. The idea of reclaiming memory for history earlier than GVT is also expressed in Figure 10, which shows IQ and OQ sections for entries with timestamp later than GVT only, and copies of EVL in SS if not older than GVT (also the rest of SS beyond GVT could be purged as irrelevant for Time Warp, but we assume here that the state trace is required for a post-simulation analysis).

Generally, a combination of fossil collection with any of the incremental/interleaved state saving schemes is recommended. Related to interleaving, however, rollback might be induced to events beyond the momentary committed GVT, with an average overhead directly proportional to. Not only that the interleaving of state recording is prohibiting fossil collection for states timestamped in the gap between GVT and the most recent saved state chronologically before GVT, it is also contraproductive to GVT computation which is comparably more expensive than state saving as will be seen soon.

### 2.2.7 Freeing Memory by Returning Messages

Previous strategies (interleaved, incremental state saving as well as fossil collection) are merely able to reduce the chance of memory exhaustion, but cannot actually prevent such situations from occurring. In cases where memory is already completely allocated, only additional techniques, mostly based on returning messages to senders or artificially initiating rollback, can help to escape from deadlocks due to waiting for free memory:

**Message Sendback** The first approach to recover from memory overflow in Time Warp was proposed by the *message sendback* mechanism by Jefferson [Jeff 85a]. Here, whenever the system runs out of memory on the occasion of an arriving message, part or all of the space used for saving the *input history* is used to recover free memory by returning *unprocessed* input messages (not necessarily including the one just received) back to the sender and relocating the freed (local) memory. By intuition, input messages with the highest send timestamps are returned first, since it is more likely that they carry incorrect information compared to "older" input messages, and since the annihilation of their effects can be expected not to disperse as much in virtual time, thus restricting annihilation influence spheres. Related to the original definition of the Time Warp protocol which distinguishes the *send time* (ST) and *receive time* (RT) (\( \text{ST}(m) \leq \text{RT}(m) \)) of messages, only messages with \( \text{ST}(m) > \text{IVT} \) (local *future messages*) are considered for returning. An indirect effect of the sendback could also be storage release in remote LPs due to annihilation of messages triggered by the original sender’s rollback procedure.

**Gafni’s Protocol** In a message traffic study of *aggressive* and *lazy* cancellation, Gafni [Gafn 88b] notes that *past* (\( \text{RT}(m) < \text{GVT} \)) and *present* messages (\( \text{ST}(m) < \text{GVT} < \text{RT}(m) \)) and events
accumulate in IQ, OQ, SS for the two annihilation mechanisms at the same rate, pointing out also the interweaving of messages and events in memory consumption. Past messages and events can be fossil collected as soon as a new value of GVT is available. The amount of “present” messages and events present in LP; reflects the difference of IVT; to the global GVT directly expressing the asynchrony or “imbalance” of IVT progression. This fact gives an intuitive explanation of the message sendback’s attempt to balance IVT progression across LPs, i.e. intentionally rollback those LPs that have progressed IVT ahead of others. Gafni, considering this asynchrony to be exactly the source from which Time Warp can gain real execution speedup, states that IVT progression balancing is does not solve the storage overflow problem. His algorithm reclaims memory by relocating space used for saving the input or state or output history in the following way: Whether the overflow condition is raised by an arriving input message, the request to log a new state or the creation of a new output message, the element (message or event) with the largest timestamp is selected irrespective of its type.

- If it is an output message, a corresponding antimessage is sent, the element is removed from OQ and the state before sending the original message is restored. The antimessage arriving at the receiver will find its annihilation partner in the receiver’s IQ upon arrival (at least in FIFO CSs), so memory is also reclaimed in the receiver LP.
  - If it is an input message, it is removed from IQ and returned to the original sender to be annihilated with its dual in the OQ, perhaps invoking rollback there. Again also the receiver LP relocates memory.
  - If it is a state in SS, it is discarded (and will be recomputed in case of local rollback).

The desirable property of both message sendback and Gafni’s protocol is that LPs that ran out of memory can be relieved without shifting the overflow condition to another LP. So, given a certain minimum but limited amount of memory, both protocols make Time Warp “operable”.

**Cancelback** An LP simulation memory management scheme is considered to be storage optimal if it consumes \(O(M^{seq})\) constant bounded memory [Lin 91]. The worst case space complexity of Gafni’s protocol is \(O(NM^{seq}) = O(N^2)\) (irrespective of whether memory is shared or distributed), the reason for this being that it can only cancel elements within the individual LPs. Cancelback is the first optimal memory management protocol [Jeff 90], and was developed targeting Time Warp implementations on shared memory systems. As opposed to Gafni’s protocol, in Cancelback elements can be canceled in any LP; (not necessarily in the one that observed memory overflow), whereas the element selection scheme is the same. Cancelback thus allows to selectively reclaim those memory spaces that are used for the very most recent (globally seen) input-, state- or output-history records, whichever LP maintains this data. An obvious implementation of Cancelback is therefore for shared memory environments and making use of system level interrupts. A Markov chain model of Cancelback [Akyi 93] predicting speedup as the amount of available memory beyond \(M^{seq}\) is varied, revealed that even with small fractions of additional memory the protocol performs about as well as with unlimited memory. The model assumes totally symmetric workload and a constant number of messages, but is verified with empirical observations.
**Artificial Rollback** Although Cancelback theoretically solves the memory management dilemma of Time Warp since it produces correct simulations in real, limited memory environments with the same order of storage requirement as the sequential DES, it has been criticized for its implementation not being straightforward, especially in distributed memory environments. Lin [Lin 91] describes a Time Warp management scheme that is in turn memory optimal (there exists a shared memory implementation of Time Warp with space complexity \(O(M^{*e})\)), but has a simpler implementation. Lin’s protocol is called *artificial rollback* for provoking the rollback procedure not only for the purpose of *loc*-violation restoration, but also for its side effect of reclaiming memory (since rollback as such does not affect operational correctness of Time Warp, it can also be invoked *artificially*, i.e., even in the absence of a straggler). Equivalent to Cancelback in effect (cancelling an element generated by LP\(_j\) from IQ\(_i\) is equivalent to a rollback in LP\(_j\), whereas cancelling an element from OQ\(_i\) or SS\(_i\) is equivalent to a rollback in LP\(_j\)), artificial rollback has a simpler implementation since the rollback procedure already available can be used together with an artificial rollback trigger causing only very little overhead. Determining, however, in which LP\(_i\) to invoke artificial rollback, to what IVT to rollback and at what instant of real time \(T\) to trigger it is not trivial (except the triggering, which can be related to the overflow condition and the failure of fossil collection). In the implementation proposed by Lin and Preiss [Lin 91], the two other issues are coupled to a processor scheduling policy in order to guarantee a certain amount of free memory (called *salvage parameter* in [Nico 94]), while following the “cancel-furthest-ahead” principle.

**Adaptive Memory Management** The *adaptive memory management* (AMM) scheme proposed by Das and Fujimoto [Das 94] attempts a combination of controlling optimism in Time Warp and an automatic adjustment of the amount of memory in order to optimize fossil collection, Cancelback and rollback overheads. Analytical performance models of Time Warp with Cancelback [Akyi 93] for homogeneous (artificial) workloads have shown that at a certain amount of available free memory fossil collection is sufficient to allocate enough memory. With a decreasing amount of available memory, absolute execution performance decreases due to more frequent cancelbacks until it becomes frozen at some point. Strong empirical evidence has been given as a support to this analytical observations. The motivation now for an *adaptive* mechanism to control memory is twofold: (i) absolute performance is supposed to have negative increments after reducing memory even further. Indeed, one would like to run Time Warp in the area of the “knee-point” of absolute performance. A successive adaptation to that performance optimal point is desired. (ii) the location of the knee might vary during the course of simulation due to the underlying simulation model. A runtime adaptation to follow movements of the knee is desired.

The AMM protocol for automatic adjustment of available storage uses a memory *flow model* that divides the available (limited) memory space \(M\) into three “pools”, \(M = M^c + M^{uc} + M^f\). \(M^c\) is the set of all memory locations used to store committed elements \((t(e) \leq GVT)\), \(M^{uc}\) is its analogy for uncommitted events (in IQ, OQ, or SS with \(t(e) > GVT\)) and \(M^f\) holds temporarily unused (free) memory. The behavior of Time Warp can now be described in terms of flows of (fixed sized) memory buffers (able to record one message or event for simplicity) from one pool into the other (Figure 14): Free memory must be allocated for every message created/sent, every state logged or any future event scheduled, causing buffer moves from \(M^f\) to \(M^{uc}\). Fossil collection

---

1For implementations in distributed memory environments, Time Warp with artificial rollback cannot guarantee a space complexity of \(O(M^{*e})\). Cancelback and Artificial Rollback in achieving the sequential DES storage complexity bound rely on the availability of a global, shared pool of (free) memory.
on the other hand returns buffers from $M^c$ as invoked upon exhaustion of $M^f$, whereas $M^c$ is being supplied by the progression of GVT. Buffers move from $M^{ac}$ to $M^f$ with each message annihilation, either incurred by rollback or by Cancelback. A Cancelback cycle is defined by two consecutive invocations of cancelback. A cycle starts where Cancelback was called due to failure of fossil collection to reclaim memory; at this point there are no buffers in $M^c$. Progression of IVT will move buffers to $M^{ac}$, rollback of IVT will occasionally return free memory, progression of GVT will deposit into $M^c$ to be depleted again by fossil collection, but tendentially the free pool will be drained, thus necessitating a new cancelback.

Time Warp can now be controlled by two (mutually dependent) parameters: (i) $\alpha$, the amount of processed but uncommitted buffers left behind after cancelback, as a parameter to control optimism; and (ii) $\beta$, the amount of buffers freed by Cancelback, as a parameter to control the cycle length. Obviously, $\alpha$ has to be chosen small enough to avoid rollback thrashing and overly aggressive memory consumption, but not too small in order to prevent rollbacks of states that are most likely to be confirmed (events on the critical path). $\beta$ should be chosen in such a way as to minimize the overhead caused by unnecessary frequent cancelback (and fossil collection) calls. The AMM protocol now by monitoring the Time Warp execution behavior during one cycle, attempts to simultaneously minimize the values of $\alpha$ and $\beta$, but respecting the constraints above. It assumes Cancelback (and fossil collection) overhead to be directly proportional to the Cancelback invocation frequency. Let $\varrho_{M^{ac}} = \frac{\epsilon_{committed}N}{T_{process}} - \varrho_{FC}$ be the rate of growth of $M^{ac}$, where $\epsilon_{committed}$ is the fraction of processed events also committed during the last cycle, $T_{process}$ is the average (real) time to process an event and $\varrho_{FC}$ is the rate of depletion of $M^{ac}$ due to fossil collection. (Estimates for the right hand side are generated from monitoring the simulation execution.) $\beta$ is then approximated by

$$\beta = (T_{cycle} - T_{CB,FC}) \varrho_{M^{ac}}$$

where $T_{CB,FC}$ is the overhead incurred by Cancelback and fossil collection in real time units, and $T_{cycle}$ is the current invocation interval. Indeed, $\alpha$ is a parameter to control the upper tolerable bound for the progression of IVT. To set $\alpha$ appropriately, AMM records by a marking mechanism whether an event was rolled-back by Cancelback. A global (across all LPs) counting mechanism lets AMM determine the number $\#(\epsilon_{cp})$ of events that should not have been rolled back by Cancelback,
since they were located on the critical path, and by that causing a definitive performance degrade. Starting now with a high parameter value for \( \alpha \) (which will give an observation \( \#(c_p) \approx 0 \)), \( \alpha \) is continuously reduced as long as \( \#(c_p) \) remains negligible. Rollback thrashing is explicitly tackled by a third mechanism that monitors \( e_{\text{committed}} \) and reduces \( \alpha \) and \( \beta \) to their halves when the decrease of \( e_{\text{committed}} \) hits a predefined threshold.

Experiments with the AMM protocol have shown that both the claimed needs can be achieved: Limiting optimism in Time Warp indirectly by controlling the rate of drain of free memory can be accomplished effectively by a dynamically adaptive mechanism. AMM adapts this rate towards the performance knee-point automatically, and adjusts it to follow dynamical movements of that point due to workloads varying (linearly) over time.

### 2.2.8 Algorithms for GVT Computation

So far, global virtual time has been assumed to be available at any instant of real time \( T \) in any LP, e.g. for fossil collection (S3.11) or in the simulation stopping criterion (S3). The definition of GVT(\( T \)) has been given in Section 2.2.6. An essential property of GVT(\( T \)) not mentioned yet is that it is nondecreasing over (real time) \( T \) and therefore can guarantee that Time Warp eventually progresses the simulation by committing intermediate simulation work. Efficient algorithms to compute GVT therefore are another foundational issue to make Time Warp “operable”.

The computation of GVT(\( T \)) (S3.10) generally is hard, such that in practice only estimates \( \hat{\text{GVT}}(T_i) \geq \text{GVT}(T_i) \) are attempted. Estimates \( \hat{\text{GVT}}(T_i) \), however, (as a necessity to be practically useful) are guaranteed to not overestimate the actual GVT(\( T_i \)) and to eventually improve past estimates.

**GVT Computations Employing a Central GVT Manager** Basically \( \hat{\text{GVT}}(T) \) can be computed by a central GVT manager broadcasting a request to all LPs for their current IVT and while collecting those values perform a \textit{min}-reduction. Clearly, the two main problems are that (i) messages in transit potentially rolling back a reported IVT are not taken into consideration, and (ii) all reported IVT(\( T_i \)) values were drawn at different real times \( T_i \). (i) can be tackled by message acknowledging and FIFO message passing in the CS, (ii) is generally approached by computing GVT using real time intervals \([T^\alpha_i, T^\epsilon_i] \) for every LP such that \( T^\alpha_i \leq T_i = T^\epsilon_i \leq T_i^\leq \) for all LPs. \( T^\epsilon \), thus is an instant of real time that happens to lie within every LP’s interval.

**Samadi’s algorithm** [Sama 85] follows the idea of GVT triggering via a central GVT manager sending out a GVT-start message to announce a new GVT computation epoch. After all LPs have prompted the request, the manager computes and broadcasts the new GVT value and completes the GVT epoch. The “message-in-transit” problem is solved by acknowledging every message, and reporting the minimum over all timestamps of unacknowledged messages in one LP’s OQ, together with the timestamp of first(EVL) (as the LP’s local GVT estimate, LGVT(\( T_i \))) to the GVT master.

\(^2\)The Critical Path of a DES is computed in terms of the (real) processing time on a certain target architecture respecting \( leq \). Traditionally, critical path analysis has been used to study the performance of distributed DES as reference to an “ideal”, fastest possible asynchronous distributed execution of the simulation model. Indeed, it has been shown that the length of the critical path is a lower bound on the execution time of any conservative protocol, but some optimistic protocols do exist (Time Warp with lazy cancellation, Time Warp with lazy rollback, Time Warp with phase decomposition, and the Chandy-Sherman Space-Time Method [Jeff 91]), which can surpass the critical path. The resulting possibility of so called supercritical speedup, and as a consequence its unsuitability as an absolute lower bound reference, however, has made critical path less attractive.
An improvement of Samadi’s algorithm by **Lin and Lazowska** [Lin 90] does not acknowledge every single message. Instead, to every message a sequence number is piggybacked, such that \( L P_i \) can identify missing messages as gaps in the arriving sequence numbers. Upon receipt of a control message, the protocol sends out to (all) \( L P_j \) the smallest sequence number still demanded from \( L P_j \) as an implicit acknowledgement of all the previous messages with a smaller sequence number. \( L P_j \) receiving smallest sequence numbers from other \( L P \)s can determine the messages still in transit and compute a lower bound on their timestamps.

To reduce communication complexity, **Bellenot’s algorithm** [Bell 90] embeds GLP in a *Message Routing Graph* MRG, which is mainly a composition of two binary trees with arcs interconnecting their leaves. The MRG for a GLP with \( N = 10 \) LPs e.g. would be a three level binary tree mirrored along its four node leaf base (a MRG construction procedure for arbitrary \( N \) is given in [Bell 90]). The algorithm efficiently utilizes the static MRG topology and operates in three steps:

1. **(MRG forward phase)** \( L P_0 \) (GVT manager) sends a GVT-*start* to the (one or) two successor LPs on the MRG. Once an \( L P_i \) has received GVT-*starts* from each successor, it sends a GVT-*start* in the way as \( L P_0 \) did. Every GVT-*start* in this phase defines \( T_i^\geq \) for the traversed \( L P_i \).

2. **(MRG backward phase)** The arrival of GVT-*start* messages at the last node in MRG (\( L P_N \)) defines \( T_N^\leq = T^* \). Now, starting from \( L P_N \), GVT-*lvt* messages are propagated to \( L P_0 \) traversing MRG in the opposite direction; \( T_i^\leq \) is defined for every \( L P_i \). Note that \( L P_i \) propagates “back” as an estimate the minimum of IVT\( _i \) and the estimates received. When \( L P_0 \) receives GVT-*lvt* from its child LPs in the MRG, it can, with IVT\( _0 \), determine the new estimate \( \overline{\text{GVT}}(T^*) \) as the minimum over all received estimates and IVT\( _0 \).

3. **(broadcast GVT phase)** \( \overline{\text{GVT}}(T^*) \) is now propagated along the MRG.

Bellenot’s algorithm sends less than \( 4N \) messages and uses overall \( O(\log(N)) \) time per GVT prediction epoch after an \( O(\log(N)) \) time for the initial MRG embedding. It requires a FIFO, fault free CS.

The **passive response GVT** (p\( \text{GVT} \)) algorithm [DSou 94] copes with faulty communication channels, while at the same time relaxing (i) the FIFO requirement to CS and (ii) the “centralized invocation” of the GVT computation. The latter is important since if GVT advancement is made only upon the invocation by the GVT manager, GVT cycles due to message propagation delays can become unnecessarily long in real time. Moreover, frequent invocations can make GVT computations a severe performance bottleneck due to overwhelming communication load, whereas (argued in terms of simulated time) infrequent invocations causing lags in event commitment bears the danger of memory exhaustion due to delaying fossil collection overly long. An LP-initiated GVT estimation is proposed, that leaves it to individual LPs to determine when to report new GVT information to the GVT manager. Every LP in one GVT epoch holds the GVT estimate from the previous epoch as broadcasted by the GVT master. Besides this, it locally maintains a GVT progress history, that allows each LP to individually determine when a new local GVT estimate (\( \text{LGVTT} \)) should be reported to the manager. The algorithms executed by the GVT manager and the respective \( L P_i \)’s are described as follows:
Upon receipt of LGVT, determine new estimate GVT'. If GVT' > GVT then
(2) recompute the k-sample average GVT increment as
\[ \Delta_{GVT} = \frac{1}{k} \sum_{j=n-k}^{n} \Delta_{GVT,j} \]
where \( \Delta_{GVT,j} \) is the j-th GVT increment out of a history of k observations, and
(3) broadcast the tuple \((GVT', \Delta_{GVT})\) to all LPs.

It is clearly seen that a linear predictor of the GVT increment per unit of real time is used to trigger the reporting to the manager. The receipt of a straggler in LP_i with \( ts(m_j) \leq LGVT \) naturally requires immediate reporting to the manager, even before the straggler is acknowledged itself.

A key performance improvement of pGVT is that LPs simulating along the critical path will more frequently report GVT information than others (which do not have as great of a chance to improve GVT), i.e. communication resources are consumed for the targeted purpose rather than wasted for weak contributions to GVT progression.

Distributed GVT Computation

A distributed GVT estimation procedure does not rely on the availability of common memory shared among LPs, neither is a centralized GVT manager required. Although distributed snapshot algorithms [Chan 85] find a straightforward application, solutions more efficient than message acknowledging, the delaying of sending event messages while awaiting control messages or piggybacking control information onto event messages are desired. Mattern [Matt 93] uses a "parallel" distributed snapshot algorithm to approximate GVT, that is not related to any specific control topology like a ring or the MRG topology. Moreover, it does not rely on FIFO channels.

To describe the basics of Mattern’s algorithm distinguish external events \( ee_i \in EE \) as either being send events \( se_i \in SE \) or receive events \( re_i \in RE \). The set of events \( E \) in the distributed simulation is thus the union of the set of internal events \( IE \) and the set of external events \( EE = SE \cup RE \). Both internal (\( ie_i \in IE \)) and external events (\( ee_i \in EE \)) can potentially change the state of the CI in some LP (IQ, OQ, SS, etc.), but only events \( ee_i \) can change the state of CS, i.e. the number of messages in transit. Let further be ‘→’ Lamport’s happens before relation [Lamp 78] defining a partial ordering of \( e \in E \) as follows:

1. if \( e, e' \in IE \subseteq EE \) and \( e' \) is the next after \( e \), then \( e \rightarrow e' \)
2. if \( e \in SE \) and \( e' \in RE \) is the corresponding receive event, the \( e \rightarrow e' \)
A consistent cut is now defined as $C \subseteq E$ such that
\[ (e' \in C) \land (e \rightarrow e') \Rightarrow (e \in C). \]

This means that a consistent cut separates event occurrences in LPs to belong either to the simulations past or its future. Figure 15 illustrates a consistent cut $C$, whereas $C'$ is inconsistent due to $e' \in C'$, $e \rightarrow e'$ but $e \not\in C'$ (cut events are pseudo events representing the instants where a cut crosses the time line of an LP and have no correspondence in the simulation). A cut $C'$ is later than a cut $C$ if $C \subseteq C'$, i.e., the cut line of $C'$ can be drawn right to the one for $C$. The global state of a cut can now be seen as the local state of every LP, i.e., all the event occurrences recorded in IQ, OQ, and SS up until the cut line, and the state of the channels $ch_{i,j} \neq j$ for which there exist messages in transit from the past of LP $i$ into the future of LP $j$ at the time instant of the corresponding cut event (note that a consistent cut can always be drawn as a vertical (straight) line after rearranging the events without changing their relative positions).

Matterns GVT approximation is based on the computation of two cuts $C$ and $C'$, $C'$ being later than $C$. For the computation of a single cut $C'$ the following snapshot algorithm is proposed:

(1) Every LP is colored white initially, and one LP$_{init}$ initiates the snapshot algorithm by broadcasting red control messages to all other LP$_j$ ($i \neq j$). LP$_{init}$ immediately turns to red. For all further steps, white (red) LPs can only send white (red) messages, and a white (red) message received by a white (red) LP does not change the LP’s color.

(2) Once a white LP$_i$ receives a red message it takes a local snapshot $\Sigma_i(C')$ representing its state right before the receipt of that message, and turns to red.

(3) Whenever a red LP$_i$ receives a white message, it sends a copy of it, together with its local snapshot $\Sigma_i(C')$ (containing LVT$_i(C')$) to LP$_{init}$. (White messages received by a red LP are exactly the ones considered as “in transit”.)

(3) if $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$

Figure 15: Matterns GVT Approximation Using 2 Contiguous Cuts $C$ and $C'$
(4) After LP;init has received all \( \Sigma_i(C') \) (including the respective IVT;';s) and the last copy of all \textquoteleft in transit\textquoteright messages, it can determine \( C' \) (i.e. the union of all \( \Sigma_i(C') \)). (Determination of when the last copy of \textquoteleft in transit\textquoteright messages has been received itself requires the use of \textit{distributed termination algorithm}.)

Note that the notion of a \textit{local snapshot} \( \Sigma_i(C') \) here is related to the cut \( C' \), as opposed to its relation to real time in Section 2.2.6. All \( \Sigma_i(C') \)'s are drawn at different real times by the LPs, but are all related to the same \textit{cut}. We can therefore also not follow the idea of constructing a global snapshot as \( \Sigma(T) = \bigcup_{i=1}^{N} \Sigma_i(T) \cup CS(T) \) by combining all \( \Sigma_i(T) \) and identifying \( CS(T) \), which would then trivially let us compute \( GVT(T) \). Nevertheless, Mattern's algorithm can be seen as an analogy: all local snapshots \( \Sigma_i(C') \) are related to \( C' \) and the motivation is to determine a global snapshot \( \Sigma(C') \) related to \( C' \), however the state of the communication system \( CS(C') \) related to \( C' \) is not known. Some additional reasoning about the messages \textquoteleft in transit\textquoteright at cut \( C' \) is necessary. The algorithm avoids an explicit computation of \( CS(C') \), by assuming the availability of a previous cut \( C \) (\( C' \) is later than \( C \)) that isolates an epoch (of virtual time) between \( C \) and \( C' \) that guarantees certain conditions on the state of \( CS(C') \).

Algorithmically this means, that for the computation of a new GVT estimate \textit{along a \textquoteleft future\textquoteright cut \( C' \)} given the current cut \( C \), \( C' \) has to be computed following the algorithm above. Determining the minimum of all local IVT;';s from the \( \Sigma_i(C') \)'s is trivial. To determine the minimum timestamp of all the message \textquoteleft in transit\textquoteright-copy at \( C' \) (i.e. messages crossing \( C' \) in forward direction; messages crossing \( C' \) in backward direction can simply be ignored since they do not harm GVT computation), \( C' \) is moved forward as far to the right of \( C \) as is necessary to guarantee that no message crossing \( C' \) originates before \( C \), i.e. no message crosses \( C \) and \( C' \) (illustrated by dashed arrows in Figure 15). A \textit{lower bound} on the timestamp of all messages crossing \( C' \) can now be easily derived by the minimum of timestamps of all messages sent in between \( C \) and \( C' \). Obviously, the closer \( C \) and \( C' \), the better the derived bound and the better the resulting GVT approximation. The \textit{parallel} snapshot and GVT computation based on the ideas above (coloring messages and LPs, and establishing a GVT estimate based on the distributed computation of two snapshots) is sketched in [Matt 93].

2.2.9 Limiting the Optimism to Time Buckets

Quite similar to the optimistic time windows approach, the \textit{Breathing Time Bucket} (BTB) protocol addresses the antimeessage dilemma which exhibits instabilities in the performance of Time Warp. BTB is an optimistic windowing mechanism with a pessimistic message sendout policy to avoid the necessity of any antimeessage by restricting potential rollback to affect only local history records (as in SRADS [Dick 90]). BTB basically processes events in \textit{time buckets} of different size as determined by the \textit{event horizon} (Figure 16). Each bucket contains the maximum amount of causally independent events which can be executed concurrently. The \textit{local event horizon} is the minimum timestamp of any new scheduled event as the consequence of the execution of events in the current bucket in some LP. The (global) event horizon EH then is the minimum over all local event horizons and defines the lower time edge of the next event bucket. Events are executed optimistically, but messages are sent out in a \textquoteleft risk free\textquoteright way, i.e. only if they conform to EH.

Two methods have been proposed to determine when the last event in one bucket has been processed, and distribution/collection of event messages generated within that bucket can be started, but both lacking an efficient (pure) software implementation: \( (i) \) (multiple) \textit{asynchronous broadcast} can be employed to exchange \textit{local} estimates of EH in order to locally determine the global EH,
This operation can overlap the bucket computation during which the CS is guaranteed to be free of event message traffic. (ii) a system wide nonblocking sync operation can be released by every LP as soon as it surpasses the local EH estimate, not hindering the LP to continue optimistically progressing computations. Once the last LP has issued the nonblocking sync, all the other LPs are interrupted and requested to send their event messages. Clearly, BTB can only work efficiently if a sufficient amount of events is processed on average in one bucket.

The Breathing Time Warp (BTW) [Stei 93] combines features of Time Warp with BTB aiming to eliminate shortcomings of the two protocols. The underlying idea again is the belief that the likelihood of an optimistically processed event being subject to a future correction decreases with the distance of its timestamp to GVT. The consequence for the protocol design is thus to release event messages with timestamps close to GVT, but delay the sendout of messages ‘distant’ from GVT. The BTB protocol operates in two modes. Every bucket cycle starts in the Time Warp mode, sending up to $M$ output messages aggressively with the hope that none of them will eventually be rolled back. $M$ is the number of consecutive messages with timestamps right after GVT. If the LP has the chance to optimistically produce more than $M$ output messages in the current bucket cycle, then BTW switches to the BTB mode, i.e. event processing continues according to BTB, but message sendout is suppressed. Should the EH be crossed in the BTB mode, then a GVT computation is triggered, followed by the invocation of fossil collection. If GVT can be improved, $M$ is adjusted accordingly.

Depending on $M$ (and the simulation model), BTW will perform somewhere between Time Warp and BTB: For simulation models with very small EH BTW will mostly remain in Time Warp mode. Frequent GVT improvements will frequently adjust $M$ and rarely allow it to be exceeded. This is indeed the desired behavior, since BTB would be overwhelmed by an increased synchronization and communication frequency in such a scenario. On the other hand, for large EH where Time Warp would overoptimistically progress due to the availability of many events in one bucket, BTW will adapt to a behavior similar to BTB since $M$ will often be exceeded and BTB will frequently be induced, thus “limiting” optimism.
2.2.10 Probabilistic Optimism

A communication interface CI that considers the CMB protocol and Time Warp as two extremes in a spectrum of possibilities to synchronize LP’s IVT advancements is the probabilistic distributed DES protocol [Fers 94c]. Let the occurrence of an event $e$, $t(e) = IVT_i$, in some LP; be causal ($\rightarrow$) for a future event $e'$ with $t(e') = IVT_j + \delta(t)$ in LP; with probability $P[e \rightarrow e']$, i.e. event $e$ with some probability changes state variables in $S_j$ with influence on $e'$. Then the CMB “block until safe-to-process”-rule appears adequate for $P[e \rightarrow e'] = 1$, and is overly pessimistic for cases $P[e \rightarrow e'] < 1$ since awaiting the decision whether ($e \rightarrow e'$) or ($e \not\rightarrow e'$) hinders the (probably correct) concurrent execution of events $e$ and $e'$ in different LPs. Clearly, if $P[e \rightarrow e'] \ll 1$ an optimistic strategy could have executed $e'$ concurrently to $e$ most of the time. This argument mainly motivates the use of an optimistic CI for simulation models with nondeterministic event causalities. On the other hand, as already seen with the discussion of rollback chains, an optimistic LP might tend towards overoptimism, i.e. optimism lacking rational justification. The probabilistic protocol aims to exploit information on the dynamic simulation behavior in order to be able to allow in every simulation step just that amount of optimism that can be justified.

To give an intuition on the justification of optimism look again at the parallel simulation of the small simulation model in Figure 7 together with the future list, and the parallel execution of lazy cancellation Time Warp in Figure 3. At the beginning of step 3 LP; at IVT = 0.56 (= IVT at the end of step 2) faces the straggler $\langle 1; P2; 0.37 \rangle$ in its IQ; the next element in LP;’s future list is 0.42. Since the effect of the straggler is in the local future of LP; i.e. $\langle T2@0.37+0.42 \rangle$, the lazy rollback strategy applies and rollback is avoided at all. The event $\langle T2@0.79 \rangle$ is executed in that step, setting IVT = 0.79, and the output message $\langle 1; P1; 0.79 \rangle$ is generated (OQ) and sent at the end of the step. Unfortunately, in step 4 a new straggler $\langle 1; P2; 0.73 \rangle$ is observed in IQ of LP; but now with the effect that at time $t = 0.73 + 0.05 < IVT = 0.79$ LP; is forced to rollback (Figure 3). Indeed, LP; in step 3 generated and sent out $\langle 1; P1; 0.79 \rangle$ without considering any information whether the implicit optimism is justified or not. If LP; would have observed that it received “on the average” one input message per step, with an “average” timestamp increment of 0.185, it might have established a hypothesis that in step 4 a message is expected to arrive with an estimated timestamp of $0.37+0.185 = 0.555$ (= timestamp of previous message + average increment). Taking this as an alarm for potential rollback, LP; could have avoided the propagation of the local optimistic simulation progression by e.g. delaying the sendout of $\langle 1; P1; 0.79 \rangle$ for one step. This is illustrated in Figure 4: LP; just takes the input message from IQ and schedules the event $\langle T2@0.79 \rangle$ in EVI, but does not process it. Instead, the execution is delayed until the hypothesis upon the next message’s timestamp is verified. The next message is $\langle 1; P1; 0.73 \rangle$, the hypothesis can be dropped, and a new event $\langle T2@0.78 \rangle$ is scheduled and processed next. Actually two rollbacks and the corresponding sending of antimessages could be avoided by applying the probabilistic simulation scheme.

<table>
<thead>
<tr>
<th>Table 4: Motivations for Probabilistic LP Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
As a more general argument, assume the timestamp of the next message $m$ carrying the external event $ee_{i+1}$ to arrive at LP$_j$ to be in the time interval $[s, t]$ ($t$ can be arbitrarily large), and that IVT$_j$ has advanced to a certain value (see Figure 17). External events have been received in the past and respected chronologically, some of the internal events are already scheduled for LP$_j$’s future. Let the probability of the actual timestamp $t$ of the forthcoming message be $P[t = x] = \frac{1}{x - s}$ $\forall x \in [s, t]$, i.e., the next external event occurrence is equally likely for all points in $[s, t]$. The CMB protocol could have safely simulated all the internal events $t(i\epsilon_k) < t(ee_{i+1})$, but will block at IVT $= s$. Under the assumptions made above, however, by blocking, CMB fails to use a $(1 - \alpha)$ chance to produce useful simulation work by progressing over $s$ and executing internal events in the time interval $[s, s + \alpha(t - s)]$. Again an overoptimism is identified for CMB, at least for small values of $\alpha$. Time Warp on the other hand would proceed simulating internal events even after surpassing $t$. This is overly optimistic in the sense that every internal event with $t(i\epsilon_k) > t$ will definitely (with probability of 1) have to be invalidated.

The probabilistic protocol appears to be a performance efficient compromise between the two classical approaches. As opposed to CMB, it allows trespassing of $s$ and progressing simulation up until $\hat{t} = ee_i + \hat{\Delta}(\delta_1, \delta_2, \ldots, \delta_n)$, $s < \hat{t} < t$, where $\hat{\Delta}(\Delta)$ is an estimate based on the differences $\delta_k = t(ee_{i-k+1}) - t(ee_{i-k})$ of the observed arrival instants $A = (t(ee_{i-n+1}), t(ee_{i-n+2}), \ldots, t(ee_i))$, $n$ being the size of the (history) observation window. Compared to Time Warp it prevents from overoptimistically diffusing incorrect simulations too far ahead into the simulated future, and thus avoids unnecessary communication overhead due to rollbacks and consequential rollback chains.

The message arrival patterns observed are used to adapt the LP to a synchronization behavior that is the best tradeoff among blocking and optimistically progressing with respect to the parallelism inherent to the simulation model, by conditioning the execution of $S3.2$ through $S3.11$ in the algorithm in Figure 11 to a probabilistic “throttle”. Assume that $\Delta_i$ = $(t(ee_{k-n+1}), t(ee_{k-n+2}), \ldots, t(ee_k))$ is maintained in LP$_j$ for every (input) channel $ch_{i,j}$, and that for $ch_{i,j}$ $\hat{t}_i$ is estimated with some confidence $0 \leq \zeta(\hat{t}_i) \leq 1$. The probabilistic protocol is then obtained by replacing $S3.2 - S3.11$ in Figure 11 with:

Figure 17: Motivation for the Probabilistic DDES
Figure 18: Delay Probabilities related to Levels of Forecast Confidence

\[ S3.2' = \begin{cases} 
\text{execute } S3.2 - S3.11 & \text{with probability } \frac{1}{1 + e^{-\gamma \zeta(\hat{t})(LVT-\hat{t})}} \\
\text{skip } S3.2 - S3.11, \text{ delay } LP_j \text{ for } \overline{\tau} & \text{with probability } \frac{1}{1 + e^{-\gamma \zeta(\hat{t})(LVT-\hat{t})}} 
\end{cases} \]

where \( \hat{t} = \min_i(\hat{t}_i) \) and \( \zeta(\hat{t}) \) the respective confidence level, \( \overline{\tau} \) is the average step execution time (in real time) and IVT is the LP's current instant of simulated time. As shown in Figure 18 assuming \( \hat{t} = 14 \), the confidence parameter \( \zeta(\hat{t}) \) together with the scaling factor \( \gamma \) (here \( \gamma = 1 \)) describes a family of probability distribution functions for delaying the SE: should there be a point estimate \( \hat{t} \) provided by some statistical method characterizing the arrival process only at a low level of confidence (\( \zeta(\hat{t}) \) small), then the delay probability directly reflects the vagueness of information justifying optimism. The more confidence (evidence) there is in the forecast, the steeper the ascent of the delay probability as IVT progresses towards \( \hat{t} \). After simulation in LP\(_j\) has surpassed the estimate \( \hat{t} \), delays are becoming more and more probable, expressing the increasing rollback hazard LP\(_j\) runs into.

The choice of the size of the observation window \( n \) as well as the selection of the forecast procedure is critical for the performance of the probabilistic protocol for two reasons: (i) the achievable prediction accuracy and (ii) the computational and space complexity of the forecast method. Generally, the larger \( n \) is chosen, the more information on the arrival history is available in the statistical sense. Respecting much of the arrival history will at least theoretically give a higher prediction precision, but will in turn consume more memory space. Intuitively, complex forecast methods could give “better” predictions than trivial ones, but are liable to intrude on the distributed simulation protocol with a nonacceptable amount of computational resource consumption. Therefore, incremental forecast methods of low memory complexity are recommended, i.e. procedures where \( \hat{t}_{i+1} \) based on \( \hat{\Delta}(\hat{\delta}_1, \hat{\delta}_2, \ldots, \hat{\delta}_{n+1}) \) can be computed from \( \hat{t}_i \) based on \( \hat{\Delta}(\hat{\delta}_1, \hat{\delta}_2, \ldots, \hat{\delta}_n) \) in \( O(\epsilon) \) instead of \( O(cn) \) time. Taking, for example, the observed mean \( \hat{\Delta}_i = \frac{1}{n} \sum_{j=1}^{n} \hat{\delta}_j \) (without imposing any
behavior of the protocol for simulation models with CMB, while for models with messages, and can even adapt to a transient behavior of the simulated model. The exponential smoothing of the observation vector by a smoothing factor \( \alpha \) (\( 1 - \alpha < 1 \)):

\[
\hat{\Delta}_{n+1} = \frac{n\hat{\Delta}_n + \delta_{n+1}}{n+1}.
\]

A possibility to weight recent history more heavily than past history could be an exponential smoothing of the observation vector by a smoothing factor \( \alpha \) (\( 1 - \alpha < 1 \)):

\[
\hat{\Delta}_{n+1} = \sum_{i=1}^{n} \alpha(1 - \alpha)^{i-1} \delta_{n+1-i}.
\]

\( \hat{\Delta} \) in this case has the incremental form:

\[
\hat{\Delta}_{n+1} = \alpha \delta_{n+1} + (1 - \alpha) \hat{\Delta}_n
\]

The arrival process of messages via a channel could also be considered to originate from an underlying but unknown stochastic process. Autoregressive moving average (ARMA) process models [Broc 91] are a reasonable method to characterize that process by the relationship among a series of empirical non-independent observations \( \{X_i\} = (X_1, X_2, \ldots, X_n) \) (in our case \( \{\delta_1, \delta_2, \ldots, \delta_n\} \)). Assuming \( \{X_i\} \) is already a centered series (i.e., transformed with respect to the series mean \( \mu \), \( X_i = \delta_i - \mu \)), then

\[
X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \ldots + \phi_p X_{t-p} + \epsilon_t
\]

is a pure autoregressive process of order \( p \) (AR\( [p] \)) with \( \epsilon_t \) being a sequence of (independent, identically distributed) white noise random disturbances. \( X_t \) is usually called the centered response, and \( \phi_i \) are parameters that can be estimated from the realizations in various different ways [Broc 91], e.g., maximum likelihood or the (recursive) Yule-Walker method. \( X_t = \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} \) is a pure moving average process of order \( q \) (MA\( [q] \)) with \( \text{E}(\epsilon_t) = 0 \), \( \text{Var}(\epsilon_t) = \sigma^2 \) and \( \text{E}(X_t) = 0 \). A (mixed) process ARMA\( [p, q] \) is now defined as

\[
X_t = \sum_{i=1}^{p} \phi_i X_{t-i} + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}
\]

Stationary ARMA\( [p, q] \) processes are able to explain short term trends and even cycles in the observation pattern, thus characterizing the transient behavior of message arrivals (in the case the simulation induces phases of message arrival patterns) to some extent. Several methods are available for ARMA\( [p, q] \) processes to forecast \( X_{t+1} \) from \( \{X_t\} \). An incremental way, for example, is the Durbin-Levinson method for one-step best linear predictions. Non-stationary series \( X_{t+1} \) can be treated with ARIMA\( [p, d, q] \) processes, \( d \) in this case denoting the differencing order, i.e., the number of differencing transformations required to induce stationarity for the non-stationary ARIMA process (ARIMA\( [p, 0, q] \sim \text{ARMA}[p, q] \)). Besides the basic statistical methods described above, also (nonlinear) mechanisms developed in control theory, machine learning and neural network simulations as well as hidden markov models could be used to control optimism in the probabilistic protocol.

The major strength of the probabilistic protocol is that the optimism of Time Warp can be automatically controlled by the model parallelism available as expressed by the likelihood of future messages, and can even adapt to a transient behavior of the simulated model. The asymptotic behavior of the protocol for simulation models with \( P[\epsilon \rightarrow e'] \approx 1 \) (for all pairs \( (\epsilon, e') \)) is close to CMB, while for models with \( P[\epsilon \rightarrow e'] \approx 0 \) it is arbitrarily close to (a throttled) Time Warp.
3 Conservative vs. Optimistic Protocols?

The question on the relative qualities of conservative and optimistic protocols has often been raised. General rules of superiority cannot be formulated, since performance – due to a very high degree of interweaving of influencing factors – cannot be sufficiently characterized by models, although exceptions do exist [Akyi 93]. Even full implementations often prohibit performance comparisons if different implementation strategies were followed (a performance comparable implementation design is worked out in [Chio 93c]) or different target platforms were selected. Performance influences on behalf of the platform come from the hardware as such (communication/computation speed ratio), the communication model (FIFO, routing strategy, interconnection network topology, possibilities of broadcast/multicast operations, etc.) and the synchronization model (global control unit, asynchronous distributed memories, shared variables, etc.) making protocols widely uncomparable across platforms. Protocol specific optimizations, i.e. optimizations in one protocol that do not have a counterpart in the other scheme (e.g. lazy cancellation) hinder even more a “fair” comparison. We therefore separate arguments in a more or less rough way:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Conservative (CMB)</th>
<th>Optimistic (Time Warp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operational</td>
<td><em>lcc</em> violation is strictly avoided; only <em>safe</em> (“good”) events are processed</td>
<td><em>lcc</em> violation occur, but recovers when detected (immediately or in the future); processes “good” and “bad” events, eventually commits good ones, cancels bad ones</td>
</tr>
<tr>
<td>Principle</td>
<td><strong>Synchronization</strong></td>
<td><strong>Synchronization</strong></td>
</tr>
<tr>
<td></td>
<td>synchronization mechanism is processor blocking; as a consequence prone to deadlock situations (deadlock is a protocol intrinsic, not a resource contention problem); deadlock prevention protocols based on nullmessages are liable to severe communication overheads; deadlock detection and recovery protocols mostly rely on a centralized deadlock manager</td>
<td>synchronization mechanism is <em>rollback</em> (of simulated time); consequential remote annihilation mechanisms are liable to severe communication overheads; cascades of rollbacks that will eventually terminate can burden execution performance and memory utilization</td>
</tr>
<tr>
<td>Parallelism</td>
<td>model parallelism cannot be fully exploited; if causalities are probable but seldom, protocol behaves overly pessimistic</td>
<td>model parallelism is fully exploitable; if causalities are probable but frequent, the Time Warp can gain most of the time</td>
</tr>
<tr>
<td>Lookahead</td>
<td>necessary to make CMB operable, essential for performance</td>
<td>Time Warp does not rely on any model related lookahead information, but lookahead can be used to optimize the protocol</td>
</tr>
<tr>
<td><strong>Balance</strong></td>
<td>CMB performs well as long as all static channels are equally utilized; large dispersion of events in space and time is not bothersome.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td><strong>GVT</strong></td>
<td>Time Warp performs well if average LVT progression is “balanced” among all LPs; space time dispersion of events can degrade performance.</td>
<td></td>
</tr>
<tr>
<td><strong>States</strong></td>
<td>relies on explicit GVT which is generally hard to compute; centralized GVT manager algorithms are liable to communication bottlenecks if no hardware support; distributed GVT algorithms impose high communication overhead and seem less effective.</td>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>conservative memory utilization copes with simulation models having “arbitrarily” large state spaces.</td>
<td></td>
</tr>
<tr>
<td><strong>Messages and Communication</strong></td>
<td>conservative memory consumption (as a consequence of the scheme).</td>
<td></td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>aggressive memory consumption; state saving overhead; fossil collection requires efficient and frequent GVT computation to be effective; complex memory management schemes necessary to prevent memory exhaustion.</td>
<td></td>
</tr>
<tr>
<td><strong>Messages and Communication</strong></td>
<td>timestamp order arrival of messages and event processing mandatory; strict separation of input channels required; static LP interconnection channel topology;</td>
<td></td>
</tr>
<tr>
<td><strong>Messages and Communication</strong></td>
<td>messages can arrive out of chronological order, but must be executed in timestamp order; one single input queue; no static communication topology; no need to receive messages in sending order (FIFO), can thus be used on more general hardware platforms.</td>
<td></td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>straightforward to implement; simple control and data structures.</td>
<td></td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>hard to implement and debug; simple data structures, but complex data manipulations and control structures; “tricky” implementations of control flow (interrupts) and memory organization essential; several performance influencing implementation optimizations possible.</td>
<td></td>
</tr>
</tbody>
</table>
Performance mainly relies on deadlock management strategy; computational and communication overhead per event is small on average; protocol in favor of “fine grain” simulation models; no general performance statement possible

mainly relies on excessive optimism control and strategy to manage memory consumption; computational and communication overhead per event is high on average; protocol in favor of “large grain” simulation models; no general performance statement possible

Indeed, a protocol comparison is of less practical importance than motivated by much of the analysis literature. Issues with more practical relevance [Fuji 93] are the design of simulation languages and the development of tools to support a simulation model description independently of the sequential, parallel or distributed DES algorithm or protocol to execute it [Bagr 94], and to automate the parallelization process as far as possible. The latter is closely related to the problem of partitioning the simulation model into regions [Chio 93b] which can be conducted at least semi-automatically if model specifications are made in a formalisms abstract enough to support a structural analysis (e.g. Petri Nets) [Chio 93a, Fers 94b, Fers 94a]. The management and balancing of dynamic distributed simulation workloads is becoming more and more important with the shift from parallel processors and supercomputers to distributed computing environments (powerful workstations interconnected with high speed networks and switches) as the preferred target architecture. In this context, protocols with the possibility of dynamic LP creation and migration (dynamic rescheduling) will have to be developed. Opposed to the approaches that “virtualize” time as presented in this work, the “virtualization” of space ambitiously studied at this time [Luba 93] promises a shift of conventional parallel and distributed simulation paradigms. Further challenges are seen in the hierarchical combination [Raja 93] or even the uniformization of protocols [Bagr 91], the uniformization of continuous and discrete event simulation, the integration of real time constraints into protocols (distributed interactive simulation), etc.

Parallel and distributed simulation over the one and a half decade of its existence has turned out to be more foundational than merely exercising on the duality of Lamport’s logical clock problem. Today’s availability of parallel and distributed computing and communication technology has given relevance to the field that could not have been foreseen in its early days. Indeed, parallel and distributed simulation protocol research has just started to ferment developments in computer science disciplines other than “simulation” in the classical sense. For example, simulated executions of SIMD programs in asynchroneous environments can accelerate their execution [Shen 92], and parallel simulations executing parallel programs with message passing communication have already been shown to be possible [Dick 94]. Other work has shown that an intrusion fire monitoring and trace collection of distributed memory parallel program executions is possible by superimposing the execution with a distributed DES protocol [Turn 93]. The difficult problem of debugging parallel programs finds a high chance to be tackled by similar ideas.
4 Sources of Literature and Further Reading

Comprehensive overviews on the field of parallel and distributed simulation are the surveys by Richter and Walrand [Rich 89], Fujimoto [Fuji 90], and most recently Nicol and Fujimoto [Nico 94]. The primary reading for Time Warp is [Jeff 85a], for conservative protocols it is [Misr 86]. The most relevant literature appears in the frame of the

- Workshop on Parallel and Distributed Simulation (PADS),

formerly (while being held as part of the SCS Multiconference) published as the Proceedings of the SCS Multiconference on Distributed Simulation. PADS’s primary focus is on the development and analysis of new protocols, recently also on viability studies, successful applications, tools and development environments. Conferences and workshops that have published application, analysis, performance, implementation and comparison related studies are

- the annual Winter Simulation Conference (WSC),
- the ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems,
- the Annual Simulation Symposium, and
- a variety of conferences and workshops related to parallel and distributed processing.

Outstanding contributions can be found in the following periodicals:

- ACM Trans. on Modeling and Computer Simulation (TOMACS)
- IEEE Trans. on Parallel and Distributed Systems
- Journal of Parallel and Distributed Computing
- International Journal in Computer Simulation

46
References


48


50


