Parallel and Distributed Simulation of Discrete Event Systems

contributed to the:
Handbook of Parallel and Distributed Computing
McGraw-Hill
1995

Alois Ferscha

Institut für Angewandte Informatik, University of Vienna
Lenaugasse 2/8, A-1080 Vienna, AUSTRIA
Email: ferscha@ani.univie.ac.at

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.

CR Categories and Subject Descriptors: C.1.0 [**Processor Architectures:**] General; C.2 [**Computer Communication Networks:**] Distributed Systems — Distributed Applications; C.4 [**Computer Systems Organization:**] Performance of Systems — Modeling techniques; D.4.1 [**Operating Systems:**] Process Management — Concurrency, Deadlocks, Synchronization; I.6.0 [**Simulation and Modeling:**] General; I.6.8 [**Simulation and Modeling:**] Types of Simulation – Distributed, Parallel

General Terms: Algorithms, Performance

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

Contents

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.

2 Simulation Principles

2.1 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 sill subsequently only consider discrete simulation.

2.2 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 Δ, or in other words the, observation of the simulated dynamic system is discretized by unitary time intervals. The choice of Δ 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,

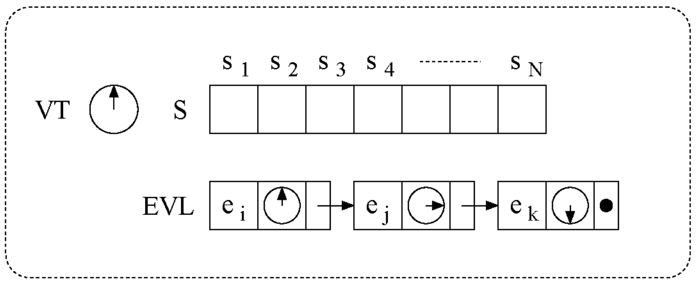


Figure 1: Simulation Engine for Discrete Event Simulation

both of which can be assembled concurrently. Part A1 requires a single assembly step, whereas A2 takes a nonpredictable amount of assemblies. 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 postprocessing 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_i's output places. The initial state of the system is represented by the marking of the PN where place $p1$ 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 occurences 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

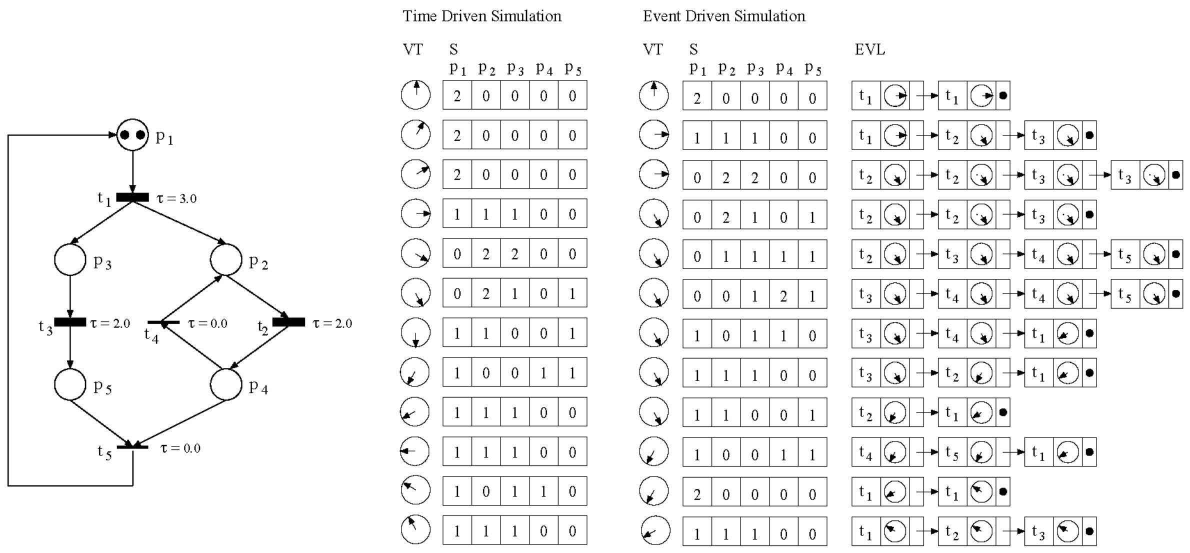

Time Driven Simulation
Event Driven Simulation
VT
S
EVL
p1 p2 p3 p4 p5
τ = 3.0
τ = 2.0
τ = 0.0
t1 t2 t3 t4 t5
2 0 0 0 0 | 2 0 0 0 0 | t1 t1
2 0 0 0 0 | 1 1 1 0 0 | t1 t2 t3
2 0 0 0 0 | 0 2 2 0 0 | t2 t2 t3 t3
1 1 1 0 0 | 0 2 1 0 1 | t2 t2 t3
0 2 2 0 0 | 0 1 1 1 1 | t2 t3 t4 t5
0 2 1 0 1 | 0 0 1 2 1 | t3 t4 t4 t5
1 1 0 0 1 | 1 0 1 1 0 | t3 t4 t1
1 0 0 1 1 | 1 1 1 0 0 | t3 t2 t1
1 1 1 0 0 | 1 1 0 0 1 | t2 t1
1 1 1 0 0 | 1 0 0 1 1 | t4 t5 t1
1 0 1 1 0 | 2 0 0 0 0 | t1 t1
1 1 1 0 0 | 1 1 1 0 0 | t1 t2 t3

Figure 2: A Sample Simulation Model described as Timed Petri Net

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 $\langle t_2@\mathrm{VT} + \tau(t_2)\rangle$ and $\langle t_3@\mathrm{VT} + \tau(t_3)\rangle$ 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.

2.3 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 5 (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 with 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.

2.3.1 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 number 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 computational 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 simulation 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 processor in this case takes care that consistency in the event structure is preserved, i.e. prohibitis 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 parallel is at the level where events from arbitrary points of the space-time are assigned to different 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 different 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.

2.4 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.*

2.5 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

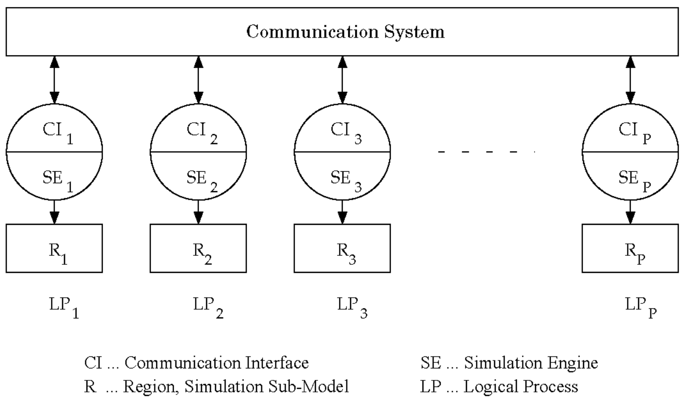


Figure 3: Architecture of a Logical Process Simulation

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$_i$ 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 occuring (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[50]: 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.

2.5.1 Synchronous LP Simulation

In a *time-stepped* LP simulation [50], 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 [62], a centralized implementation with one dedicated processor controlling the global clock is proposed. To overcome stepping the time at instances where no events are occuring, 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 [50], a structured (hierarchical) LP organization can be used [17] to determine the minimum next event time. A parallel min-reduction operation can bring this timestamp to the root of a process tree [4], which can then be propagated down the tree. Another possibility is to apply a distributed snapshot algorithm [11] 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 Δ-tick of time, called a bounded lag by Lubachevsky [41] or a Moving Time Window by [59].

2.5.2 Asynchronous LP Simulation

Asynchronous LP simulation relies on the presence of events occuring 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 [35] recognized this problem to be the inverse of Lamport's logical clock problem [36], i.e. providing clock values for events occuring in a distributed system such that all events appear ordered in logical time.

It is intuitively convincing and has been shown in [46] 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 [26]). Although sufficient, it is not always necessary to obey the *lcc*, because two events occuring 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 [22], that the potential performance improvement of an asynchronous LP simulation strategy over the time-stepped variant is at most $O(logP)$, 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 $T_{step,i} \sim exp(\lambda)$ in every LP$_i$ ($E[T_{step,i}] = \frac{1}{\lambda}$). As a consequence, the expected simulation time $E[T^{sync}]$ for a k time step synchronous simulation is $k\ E[\max_{i=1..P}\ (T_{step,i})]$ $= k\ \frac{1}{\lambda} \sum_{i=1}^{P} \frac{1}{i}\ \leq \frac{k}{\lambda} log(P)$. Relaxing now the synchronization constraint (as an asynchronous simulation would) the expected simulation time would be $E[T^{async}] = E[\max_{i=1..P}(k\ T_{step,i})] > \frac{k}{\lambda}$. We have $\lim_{k\to\infty,P\to\infty} \frac{E[T^{sync}]}{E[T^{async}]} \approx log(P)$, 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 $\frac{P}{log(P)}$. 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 $\lim_{k\to\infty,P\to\infty} \frac{E[T^{sync}]}{E[T^{async}]} \approx 2$, or intuitively with $T^{sync} \leq k\ u$ and $E[T^{async}] \geq k\frac{(l+u)}{2}$ the ratio of synchronous to asynchronous finishing times is $\frac{2}{(k\ u)(k\ (l+u))} \leq 2$, 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.

3 "Classical" LP Simulation Protocols

3.1 Conservative Logical Processes

LP simulations following a conservative strategy date back to original works by Chandy and Misra [12] and Bryant [9], and are often referred to as the Chandy-Misra-Bryant (CMB) protocols. As described by [46], in CMB causality of events across LPs is preserved by sending timestamped (external) event messages of type $\langle ee@t \rangle$, where ee denotes the event and t is a copy of LVT of the sending LP at (@) 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 LVT for which the LP has been guaranteed not to receive (external event) messages with $\mathrm{LVT} < t$ (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_i^{cons} to LP_j^{cons} (FIFO) is sufficient to guarantee that no out of chronological order message can ever arrive in any LP_i^{cons} (necessary for correctness). A conservative LP simulation can thus be seen as a set of all LPs $\mathrm{LP}^{cons} = \bigcup_k \mathrm{LP}_k^{cons}$ together with a set of directed, reliable, FIFO communication channels $\mathrm{CH} = \bigcup_{k,i\ (k \neq i)} ch_{k,i} = (\mathrm{LP}_k, \mathrm{LP}_i)$ that constitute the *Graph of Logical Processes* $\mathrm{GLP}^{cons} = (\mathrm{LP}, \mathrm{CH})$. (It is important to note, that GLP^{cons} has a *static* toplogy, which compared to optimistic protocols, prohibits dynamic (re-)scheduling of LPs in a set of physical processors. Note at the same time, that this view of a conservative simulation is based on a logical process model. A parallel simulation in order to be conservative does not necessarily need to employ this LP model, neither is the message transmission order assumption required [41].)

The communication interface CI^{cons} of an LP^{cons} on the input side maintains one input buffer $\mathrm{IB}[i]$ and a channel (or link) clock $\mathrm{CC}[i]$ for every channel $ch_{i,k} \in \mathrm{CH}$ pointing to LP_k^{cons} (Figure 4). $\mathrm{IB}[i]$ intermediately stores arriving messages in FIFO order, whereas $\mathrm{CC}[i]$ holds a copy of the timestamp of the message at the head of $\mathrm{IB}[i]$; initially $\mathrm{CC}[i]$ is set to zero. $\mathrm{LVTH} = \min_i \mathrm{CC}[i]$ is the time horizon up until which LVT is allowed to progress by simulating internal or external events, since no external event can arrive with a timestamp smaller than LVTH. 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 LVT meets LVTH. During this, SE might have

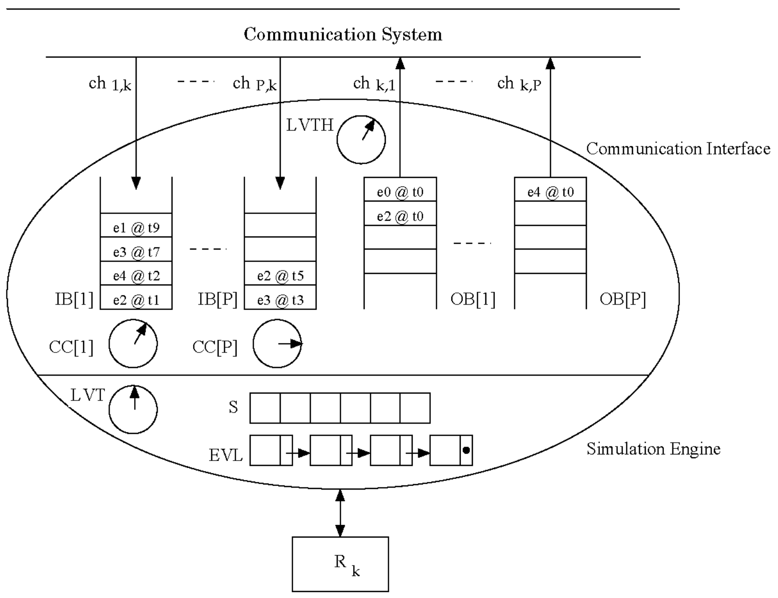


Figure 4: Architecture of a *Conservative* Logical Process

produced future events for remote LPs. For each of those, a message is constructed by adding a copy of LVT 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 $ch_{k,l} \in$ CH to subsequent LPs LP$_l$. The basic algorithm is sketched in Figure 5.

Given now that within the horizon LVTH 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.*

3.1.1 Deadlock Avoidance

Deadlock as in Figure 6 can be prevented by modifying the communication protocol based on the sending of *nullmessages* [46] of the form $\langle 0@t \rangle$, where 0 denotes a nullevent (event without effect).

```
program $LP^{cons}(\mathrm{R}_k)$
S1       LVT = 0; EVL = {}; $S$ = initialstate();
S2       for all CC[$i$] do (CC[$i$] = 0) od;
S3       for all $ie_i$ caused by $S$ do chronological_insert($\langle ie_i$@occurrence_time($ie_i$)$\rangle$, EVL) od;
S4       while LVT $\leq$ endtime do
S4.1         for all IB[$i$] do await not_empty(IB[$i$]) od;
S4.2         for all CC[$i$] do CC[$i$] = ts(first(IB[$i$])) od;
S4.3         LVTH = $\min_i$CC[$i$];
S4.4         min_channel_index = $i$ | CC[$i$] == min_channel_clock;
S4.5         if ts(first(EVL)) $\leq$ LVTH
                 then /* select first internal event*/
                     $e$ = remove_first(EVL) ;
                 else /* select first external event*/
                     $e$ = remove_first(IB[min_channel_index]);
             end if;
             /* now process the selected event */
S4.6         LVT = ts($e$);
S4.7         if not nullmessage($e$) then
S4.7.1           S = modified_by_occurrence_of($e$);
S4.7.2           for all $ie_i$ caused by $S$ do chronological_insert($\langle ie_i$@occurrence_time($ie_i$)$\rangle$, EVL) od;
S4.7.3           for all $ie_i$ preempted by $S$ do remove($ie_i$, EVL) od;
S4.7.4           for all $ee_i$ caused by $S$ do deposit($\langle ee_i$@LVT$\rangle$, corresponding(OB[$j$])) od;
             end if;
S4.11        for all empty(OB[$i$]) do deposit($\langle$0@LVT + lookahead($ch_{k,i}$)$\rangle$, OB[$i$]) od;
S4.12        for all OB[$i$] do send_out_contents(OB[$i$]) od;
         od while;
```

Figure 5: Conservative LP Simulation Algorithm Sketch.

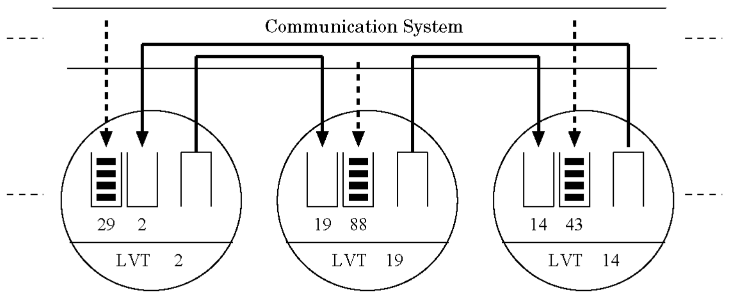


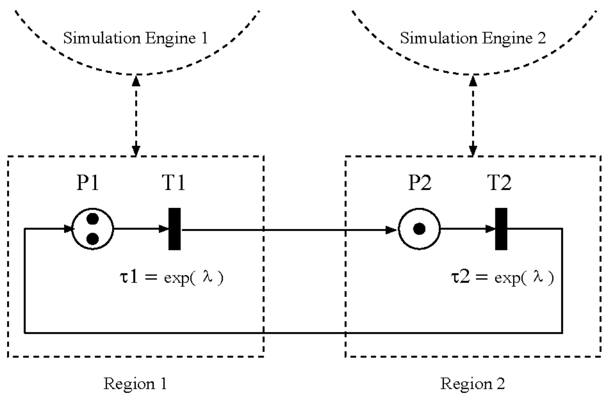
Figure 6: Deadlock and Memory Overflow

A nullmessages 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), after the LP in the middle would have broadcasted $\langle 0@19 \rangle$ to the neighboring LPs, both of them would have chance to progress their LVT up until time 19, and in turn issue new event messages expanding the LVTHs 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 [46]. An approach where additional information (essentially the routing path as observed during traversal) is attached to the nullmessage, the *carrier nullmessage protocol* [10] 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 LVT 19 could know that processing the next event will certainly increment LVT to 22, then nullmessages $\langle 0@22 \rangle$ (so called *lookahead* of 3) could have been broadcasted as further improvement on the LVTH 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 [47], the basic idea being that the simulation of a job arriving at a FCFS queue will certainly increment LVT 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.



Future Lists for λ = 2

T1	T2
0.37	0.51
0.17	0.39
0.22	0.42
0.34	0.05
0.93	0.88
0.65	0.17

Figure 7: LP Simulation of a Trivial PN with Model Parallelism

3.1.2 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 ($\tau(\text{T1}) \sim exp(0.5)$, $\tau(\text{T2}) \sim exp(0.5)$), 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 LP_1 and LP_2 are: $R_1 = (\{\text{T1}\}, \{\text{P1}\}, \{(\text{P1}, \text{T1})\}, \tau(\text{T1}) \sim exp(\lambda = 0.5))$, and $R_2 = (\{\text{T2}\}, \{\text{P2}\}, \{(\text{P2}, \text{T2})\}, \tau(\text{T2}) \sim exp(\lambda = 0.5))$.

Let the *future list* [47], 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 (*tokenmessage*), and a *timestamp* equal to the *local virtual time* LVT of the sending LP. On the other hand, the receipt of an event message (external event)

Step	VT	S	EVL	T
0	0.00	(2,1)	T1@0.17; T1@0.37; T2@0.51	—
1	0.17	(1,2)	T1@0.37; T2@0.51; T2@0.56	T1
2	0.37	(0,3)	T2@0.51; T2@0.56; T2@0.79	T1
3	0.51	(1,2)	T2@0.56; T1@0.73; T2@0.79	T2
4	0.56	(2,1)	T1@0.73; T2@0.79; T1@0.90	T2
5	0.73	(1,2)	T2@0.78; T2@0.79; T1@0.90	T1

Table 1: Sequential DES of a PN with Model Parallelism

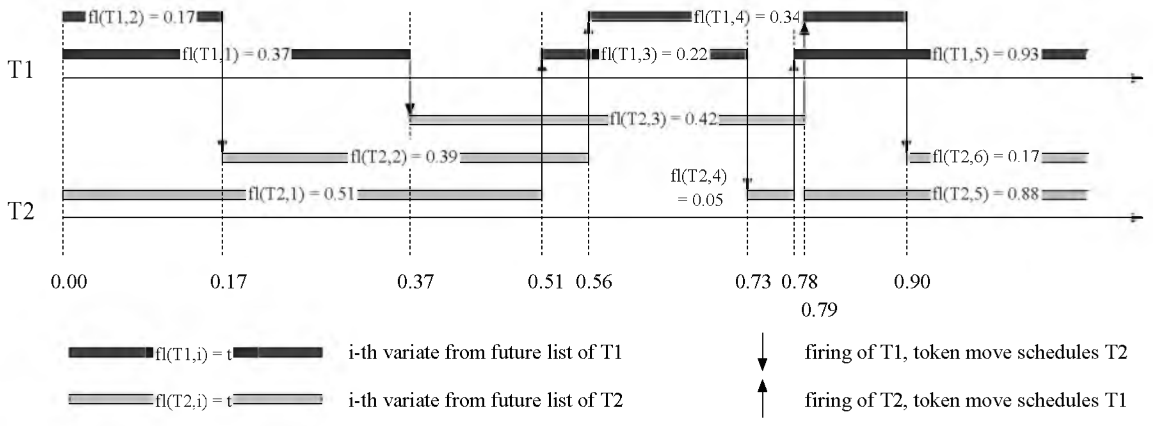


Figure 8: Model Parallelism Observed in the PN execution

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 LVT 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 P_1 and one in P_2), 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 $\min(0.51, 0.39, 0.42) = 39$ 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*

Step	LP_1						LP_2					
	IB	LVT	S_{P1}	EVL	OB	B	IB	LVT	S_{P2}	EVL	OB	B
0	—	0.00	2	T1@0.17; T1@0.37	—		—	0.00	1	T2@0.51	—	
1	—	0.00	2	T1@0.17; T1@0.37	⟨ 0; P2; 0.17 ⟩	•	—	0.00	1	T2@0.51	⟨ 0; P1; 0.39 ⟩	•
2	⟨ 0; P1; 0.39 ⟩	0.17	1	T1@0.37	⟨ 1; P2; 0.17 ⟩		⟨ 0; P2; 0.17 ⟩	0.17	1	T2@0.51	⟨ 0; P1; 0.51 ⟩	•
3	⟨ 0; P1; 0.51 ⟩	0.37	0	—	⟨ 1; P2; 0.37 ⟩		⟨ 1; P2; 0.17 ⟩	0.17	2	T2@0.51; T2@0.56	⟨ 0; P1; 0.51 ⟩	•
4	⟨ 0; P1; 0.51 ⟩	0.51	0	—	⟨ 0; P2; 0.73 ⟩	•	⟨ 1; P2; 0.37 ⟩	0.37	3	T2@0.51; T2@0.56; T2@0.79	⟨ 0; P1; 0.51 ⟩	•
5	⟨ 0; P1; 0.51 ⟩	0.51	0	—	⟨ 0; P2; 0.73 ⟩	•	⟨ 0; P2; 0.73 ⟩	0.51	2	T2@0.56; T2@1.79	⟨ 1; P1; 0.51 ⟩	
6	⟨ 1; P1; 0.51 ⟩	0.51	1	T1@0.73	⟨ 0; P2; 0.73 ⟩	•	⟨ 0; P2; 0.73 ⟩	0.56	1	T2@0.79	⟨ 1; P1; 0.56 ⟩	
7	⟨ 1; P1; 0.56 ⟩	0.56	2	T1@0.73; T1@0.90	⟨ 0; P2; 0.73 ⟩	•	⟨ 0; P2; 0.73 ⟩	0.73	1	T2@0.79	⟨ 0; P1; 0.78 ⟩	•
8	⟨ 0; P1; 0.78 ⟩	0.73	1	T1@0.90	⟨ 1; P2; 0.73 ⟩		⟨ 0; P2; 0.73 ⟩	0.73	1	T2@0.79	⟨ 0; P1; 0.78 ⟩	•

Table 2: Parallel Conservative LP Simulation of a PN with Model Parallelism

occurence of T1 must be $t(\mathrm{T1}_2) = 0.37$. The first occurrence of T2 can be either the one scheduled at 0.51, or the one invoked by the first occurence of T1 at $0.17 + 0.39 = 0.56$, or the one invoked by the second occurence of T1 at $0.37 + 0.42 = 0.78$. Clearly, the *first* occurence of T2 must be at $t(\mathrm{T2}_1) = 0.51$, and the *second* occurrence of T2 must be at $t(\mathrm{T2}_2) = 0.17 + 0.39 = 0.56$, etc. Since $\mathrm{T1}_1 \rightarrow \mathrm{T2}_2$ with $t(\mathrm{T2}_1) < t(\mathrm{T2}_2)$ and $\mathrm{T2}_1 \rightarrow \mathrm{T1}_3$ with $t(\mathrm{T1}_1) < t(\mathrm{T1}_3)$, $\mathrm{T1}_1$ and $\mathrm{T2}_1$ do not interfere with each other and can therefore be simulated independently ($\mathrm{T1}_i \rightarrow \mathrm{T2}_j$ denotes the *direct scheduling* causality of the $i-$th occurrence of T1 onto the $j-$th occurrence of T2).

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= $(\{\mathrm{LP}_1, \mathrm{LP}_2\}, \{ch_{1,2}, ch_{2,1}\})$, where the channels $ch_{1,2}$ and $ch_{2,1}$ are supposed to substitute the PN arcs (T1, P2) and (T2, P1) 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 $m = \langle 1, \mathrm{P2}, t \rangle$ from LP_1 to LP_2 on the occurrence of a firing of T1, in order to deposit 1 (first component of m) token into place P2 (second component of m) at time t (third component). The timestamp t is produced as a copy of the LVT of LP_1 at the instant of that firing of T1, that produced the token.

A CMB *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 LVTH = 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 outputchannels j. In the example we have

$$la(ch_{i,j}) = min(\ (\mathrm{LVT}_i - min_{k=1..S_i}(st_k))\ ,\ min_{k=1..(M-S_i)}\ fl_k\ )$$

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 LVT, giving the timestamps for the nullmessage $\langle$ 0; P2; 0.17 $\rangle$ sent from LP_1 to LP_2, and $\langle$ 0; P1; 0.39 $\rangle$ sent from LP_2 to LP_1. The latter, when arriving at LP_1, unblocks the SE_1, such that the first event out of EVL_1 can be processed, generating the event message $\langle$ 1; P1; 0.17 $\rangle$. 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 $\langle$ 0; P1; 0.51 $\rangle$ 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.

3.1.3 Deadlock Detection/Recovery

An alternative to the Chandy-Misra-Bryant protocol avoiding nullmessages has also been proposed by Chandy and Misra [13], 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 LVT. 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 [44] or distributed deadlock detection algorithm [14] could be used instead.

In an algorithm described by Misra [46], a special message called *marker* circulates through GLP 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 [5] 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 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 LVT 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* replys, 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 LVT equals or exceeds the requested time a *yes* is replied, whereas if the LP's LVT 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 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 received 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 [31] 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.

3.1.4 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_i [41], [48].

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 $e \in \mathrm{W}_i$, e is causally independent of any $e' \in \mathrm{W}_j$, $j \neq i$. Phase (i) is accomplished by a barrier synchronization over all LPs. In phase (ii) (*event processing*) every LP_i processes events $e \in \mathrm{W}_i$ 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_i (window identification phase). Lubachevsy [41] 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 know 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 $O(logN)$ 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.

3.1.5 The Carrier Null Message Protocol

Another approach to reduce the overwhelming amount of null messages occuring with the CMB protocol is to add more information to the null messages. The *carrier null message protocol* [10] uses nullmessages 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 nullmessages 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 $la(ch_{i,j}) = 2\ \forall i,j \in \{\text{source}, \text{join}, \text{pass}, \text{split}, \text{sink}\}$, $(i \neq j)$, except $la(ch_{\text{source,join}}) = 50$. After the first object release into LP_{join}, all LPs except $\text{LP}_{\text{source}}$ are blocked, and therefore start propagating local lookahead via nullmessages. After the propagation of (overall) 4 nullmessages all LPs beyond $\text{LP}_{\text{source}}$ have progressed LVT'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 $\langle c0, t, \mathcal{R}, la.inf \rangle$, where $c0$ is an identification as a *carrier nullmessage*, t is the timestamp, $\mathcal{R}$ 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 $\mathcal{R}$, it can be sure (but only in the paricular 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 $\text{LP}_{\text{source}}$, and thus increment the CC's and LVT's of all successors on the route in $\mathcal{R}$ 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

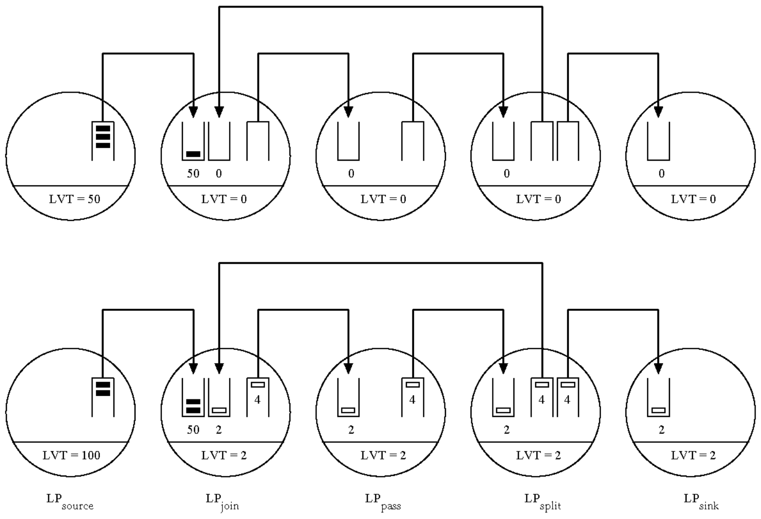


Figure 9: Motivation for Lookahead Propagation using CNM

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.

3.2 Optimistic Logical Processes

Optimistic LP simulation strategies, in contrast to conservative ones, do not strictly adhere to the local causality constraint *lcc* (see Section 2.5.2), 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 LVT) 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.

3.2.1 Time Warp

Pioneering work in optimistic LP simulation was done by Jefferson and Sowizral [34, 35] 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

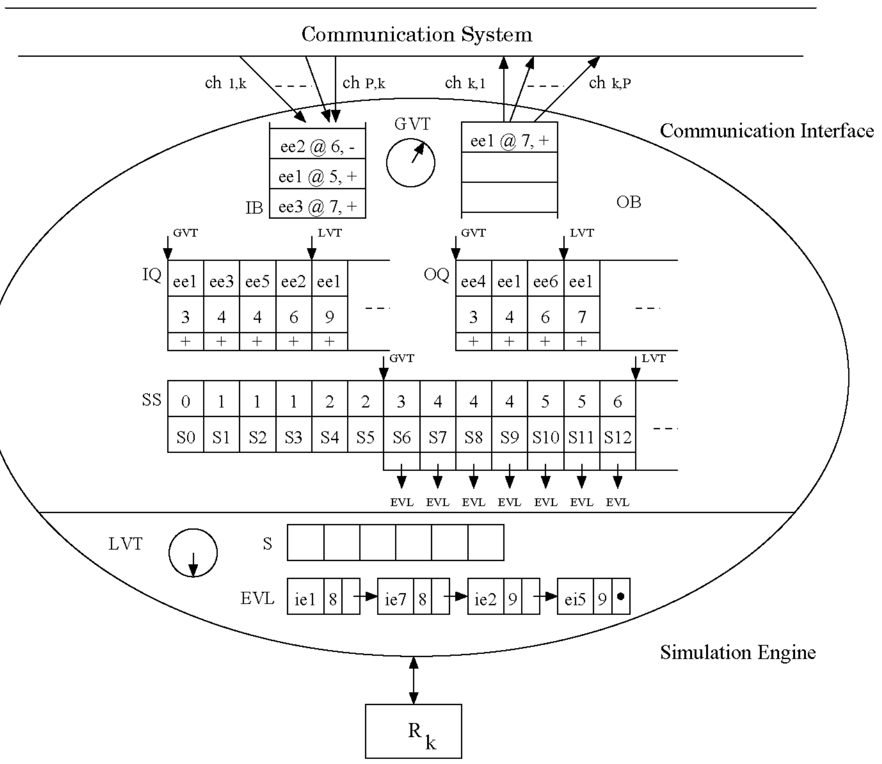


Figure 10: Architecture of an *Optimistic* Logical Process

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. 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 ($m^+ = \langle ee@t, +\rangle$), (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 ($m^- = \langle ee@t, -\rangle$) 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 LPs (*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 as was seen with LP^{cons}. The input synchronization (*Rollback and Annihilation*) and confirmation (*GVT*) part however are the key mechanisms in optimistic LP simulation.

3.2.2 Rollback and Annihilation Mechanisms

The input synchronization of LP^{opt} (rollback mechanism) relates arriving messages to the current value of the LP's LVT 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 LVT advances to that time. $\langle ee3@7,+\rangle$ in the IB in Figure 10 is an example of such an unproblematic message (LVT = 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 *lcc*-consistent state, by reconstructing S and EVL in the simulation engine from copies attached to the SS in the communication interface. Also LVT is warped back to the timestamp of the straggler message. This so far has compensated the *local* effects of the *lcc* 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.

```
program $LP^{opt}(\mathrm{R}_k)$
S1        GVT = 0; LVT = 0; EVL = {}; $S$ = initialstate();
S2        for all $ie_i$ caused by $S$ do chronological_insert($\langle ie_i$@occurrence_time($ie_i$)$\rangle$, EVL) od;
S3        while GVT $\leq$ endtime do
S3.1          for all $m \in$ IB do
S3.1.1            if ts($m$) $\leq$ LVT /* $m$ potentially affects local past */
                    then
                        if (positive($m$) and dual($m$) $\notin$ IQ) or (negative($m$) and dual($m$) $\in$ IQ)
                          then /* rollback */
                              restore_earliest_state_before(ts($m$));
                              generate_and_sendout(antimessages);
                              LVT = earliest_state_timestamp_before($m$);
                        endif;
                  endif;
                  /* irrespective of how $m$ is related to LVT */
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          LVT = ts($e$);
S3.4          $S$ = modified_by_occurrence_of($e$);
S3.5          for all $ie_i$ caused by $S$ do chronological_insert($\langle ie_i$@occurrence_time($ie_i$)$\rangle$, EVL) od;
S3.6          for all $ie_i$ preempted by $S$ do remove($ie_i$, EVL) od;
S3.7          log_new_state($\langle$LVT, $S$, copy_of(EVL)$\rangle$, SS);
S3.8          for all $ee_i$ caused by $S$ do
                  deposit($\langle ee_i$@LVT, +$\rangle$, OB);
                  chronological_insert($\langle ie_i$@LVT, +$\rangle$, 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.

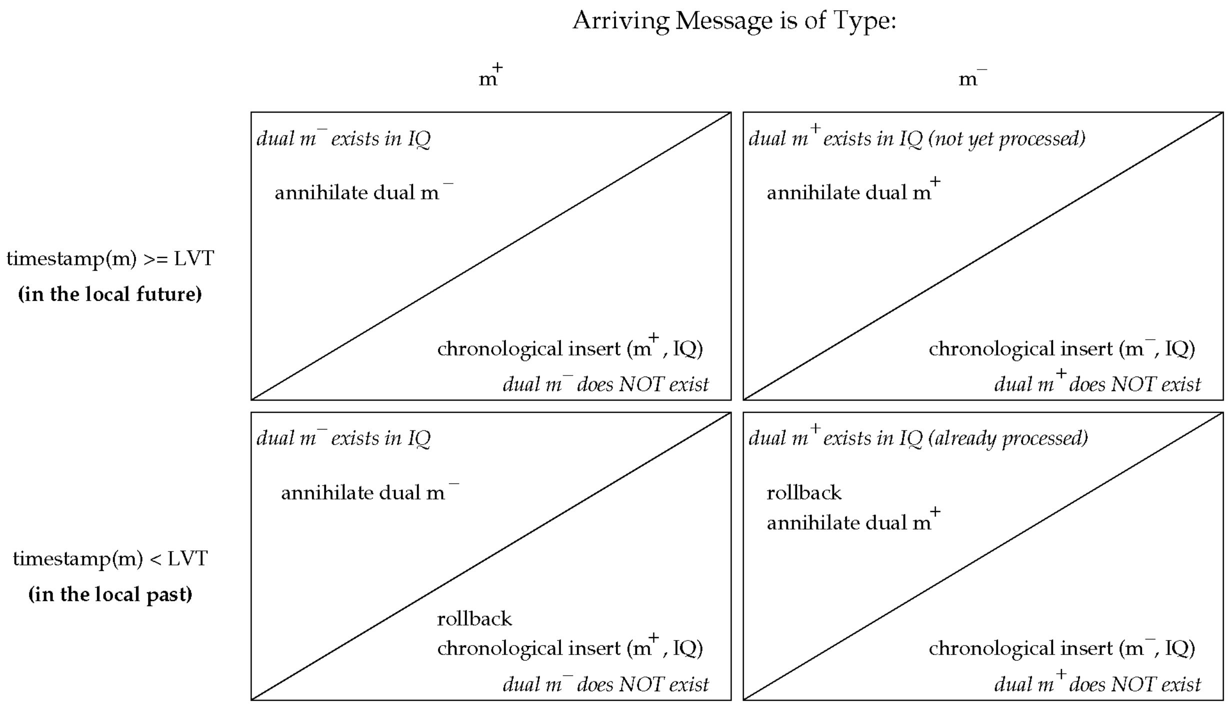


Figure 12: The Time Warp Message based Synchronization Mechanism

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 LVT) 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 (LVT, 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 antimessages 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.

Related to the possibility of rollbacks at any time of the simulation is the problem of termination detection. An alternative to the termination criterion in statement *S3* in the algorithm in Figure 11 is to introduce the timestamp ∞. An LP that completes the local simulation sets LVT$= \infty$, and every incoming message will induce a rollback. Once GVT has reached the time ∞ (i.e. LVT$= \infty$

Step	LP$_1$						LP$_2$					
	IB	LVT	S	EVL	OB	RB	IB	LVT	S	EVL	OB	RB
0	—	0.00	2	T1@0.17; T1@0.37	—		—	0.00	1	T2@0.51	—	
1	—	0.17	1	T1@0.37	⟨ 1; P2; 0.17 ⟩		—	0.51	0	—	⟨ 1; P1; 0.51 ⟩	
2	⟨ 1; P1; 0.51 ⟩	0.37	1	T1@0.73	⟨ 1; P2; 0.37 ⟩		⟨ 1; P2; 0.17 ⟩	0.56	0	—	⟨ 1; P1; 0.56 ⟩	
3	⟨ 1; P1; 0.56 ⟩	0.73	1	T1@0.90	⟨ 1; P2; 0.73 ⟩		⟨ 1; P2; 0.37 ⟩	0.79	0	—	⟨ 1; P1; 0.79 ⟩	
4	⟨ 1; P1; 0.79 ⟩	0.90	1	T1@1.72	⟨ 1; P2; 0.90 ⟩		⟨ 1; P2; 0.73 ⟩	0.73	2	T2@0.78; T2@0.79	⟨ -1; P1; 0.79 ⟩	•
5	⟨ -1; P1; 0.79 ⟩	0.90	0	—	—	•	⟨ 1; P2; 0.90 ⟩	0.78	2	T2@0.79; T2@1.78	⟨ 1; P1; 0.78 ⟩	

Table 3: Parallel Optimistic LP Simulation of a PN with Model Parallelism

for every LP), termination is detected.

Lazy Cancellation In the original Time Warp protocol as described above, an LP receiving a *straggler message* initiates sending antimessages 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 antimessage (m^-) for m^+ immediately upon receipt of a straggler. Instead, it delays its propagation until the resimulation after rollback has progressed to LVT = ts(m^+) producing $m^{+'} \neq m^+$. If the resimulation produced $m^{+'} = m^+$, no antimessage has to be sent all [28]. Lazy cancellation thus avoids unnecessary cancelling of correct messages, but has the liability of additional memory and bookkeeping overhead (potential antimessages 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 $m^+ <$ LVT is received, than obviously antimessages do not have to be sent for messages m with timestamp, $ts(m^+) \leq ts(m) < ts(m^+) + la$. Moreover, if $ts(m^+) + la \geq$ LVT 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 $\langle 1; \mathrm{P1}; 0.17 \rangle$ at LVT = 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 [33] 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 computations 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 [53] 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 α time using N processors, the other uses αN 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 LVT, *lazy re-evaluation* delays discarding entries on the state stack SS. Should the recomputation after rollback to time $t <$ LVT 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 LVT, the time before rollback occured. 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 [26]. To verify equivalence of IQ's the protocol must draw and log copies of the IQ in every state saving step (*S3.7*). 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 $ts(m^*) + la(ee) \geq$ LVT can again be overjumped, saving the computation time for the resimulation of events in between $[ts(m^*), ts(m^*) + la(ee))$. $la(ee)$ 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 [51], 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 [43] 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 Reynolds [54, 20], 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* $W(\mathrm{LP}_i, \tau)$, defined as the set of LPs that can be influenced by a message from LP_i 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_i are limited to a sphere of a computable (or selectable) radius around LP_i, 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 being provably smaller than for the standard Time Warp protocol).

3.2.3 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 [59], 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 $\mid ts(e) - ts(e') \mid < \Delta$. 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 $\min_i t_i(e), \min_i t_i(e) + \Delta$. (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 $\min_i t_i(e)$ 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 Δ 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 Hyot [6], 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 *lcc* violations in the past. In this case the LP would conclude that it progresses LVT too fast compared to the predecessor LPs and would therefore stop LVT 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.

3.2.4 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 [40], 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

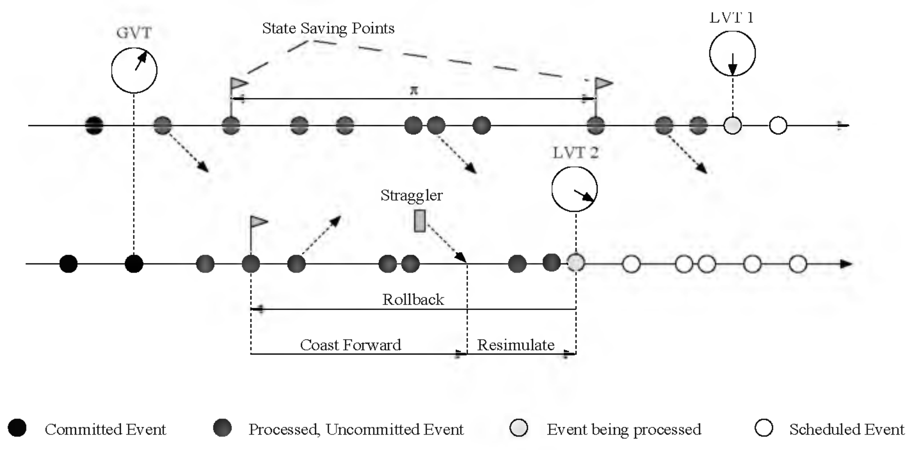


Figure 13: Interleaved State Saving

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.

3.2.5 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$_i$ that do not extensively change values over LVT 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    if (step_count modulo $\pi$) == 0 then log_new_state($\langle$LVT, $S$, copy_of(EVL)$\rangle$, SS);
```

could be used to *interleave* the continuity of saved states and thus on the average reduce the storage requirement to $\frac{1}{\pi}$ 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 [49] in reference to Lin and Lazowska) based on expected event processing time ($\epsilon = E[exec(e)]$) and state saving costs (σ), giving an optimal *interleaving factor* π^* as

$$\lfloor\sqrt{(\alpha-1)\beta}\rfloor < \pi^* < \lceil\sqrt{(2\alpha+1)\beta}\rceil$$

where α is the average number of rollbacks with $\pi = 1$ and $\beta = \frac{\sigma}{\epsilon}$. The result expresses that an overestimation of π^* 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 π does neither affect the number of rollbacks nor the number of rolled back events in [38], an algorithm is developed that, integrated into the protocol, "on-the-fly", within a few iterations, automatically adjusts π to π^*. It has been shown that some π, though increasing the rollback overhead, can reduce overall execution time.

3.2.6 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

$$\Sigma_i(T) = (\mathrm{LVT}_i(T), \mathrm{IQ}_i(T), \mathrm{SS}_i(T), \mathrm{OQ}_i(T))$$

to be a *local snapshot* of LP_i at *real time* T, i.e. LVT, IQ_i is the input queue as seen by an external observer at *real time* T, etc., and $\Sigma(T) = \bigcup_{i=1}^{N} \Sigma_i(T) \cup \mathrm{CS}(T)$ be the *global snapshot* of GLP. Further let $\mathrm{LVT}_i(T)$ 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 Σ. 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:

$$\mathrm{GVT}(T) = \min(\min_i \mathrm{LVT}_i(T), \min_{i,j;m \in UM_{i,j}(T)} ts(m))$$

It should be clear even by intuition, that at *any* (real time) T, $\mathrm{GVT}(T)$ represents the maximum lower bound to which any rollback could ever backdate LVT_i ($\forall i$). An obvious consequence is that any processed event e with $ts(e) < \mathrm{GVT}(T)$ can never (at no instant T) be rolled back, and can therefore be considered as *(irrevocably) committed* [40] (Figure 13). Further consequences (for all LP_i) are that:

(i) messages $m \in \mathrm{IQ}_i$ with $ts(m) \leq \mathrm{GVT}(T)$, as well as messages $m \in \mathrm{OQ}_i$ with $ts(m) \leq \mathrm{GVT}(T)$ are obsolete and can be discarded (from IQ, OQ) after real time T.

(ii) state variables $s \in S_i$ stored in SS_i as with $ts(s) \leq \mathrm{GVT}(T)$ 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 π. 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.

3.2.7 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 [35]. Here, whenever the system runs out of memory on the occasion of an arriving message, part or all af 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 restriciting annihilation influence spheres. Related to the original definition of the Time Warp protocol which distinguishes the *send time* (ST) and *receive time* (RT) ($\mathrm{ST}(m) \leq \mathrm{RT}(m)$) of messages, only messages with $\mathrm{ST}(m) > \mathrm{LVT}$ (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 [28] notes that *past* ($\mathrm{RT}(m) < \mathrm{GVT}$) *and present* messages ($\mathrm{ST}(m) < \mathrm{GVT} < \mathrm{RT}(m)$) 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_i reflects the difference of LVT_i to the global GVT directly expressing the asynchrony or "imbalance" of LVT progression. This fact gives an intuitive explanation of the *message sendback*'s attempt to *balance* LVT progression across LPs, i.e. intentionally rollback those LPs that have progressed LVT ahead of others. Gafni, considering this asynchrony to be exactly the source from which Time Warp can gain real execution speedup, states that LVT progression balancing is does not solve the storage overflow problem. His algorithm reclaims memory by relocating space used for saving the *input* <u>or</u> *state* <u>or</u> *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* iff it consumes $O(M^{seq})$ *constant bounded* memory [40]. 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 [32], 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$_i$ (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 [1] 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 [40] describes a Time Warp management scheme that is in turn memory *optimal* (there exists a shared memory implementation of Time Warp with space complexity $O(M^{seq})$ [1]), but has a simpler implementation. Lin's protocol is called *artificial rollback* for provoking the rollback procedure not only for the purpose of *lcc*-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$_i$), 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 LVT 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 [40], 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 [49]), while following the "cancel-furthest-ahead" principle.

Adaptive Memory Management The *adaptive memory management* (AMM) scheme proposed by Das and Fujimoto [18] attempts a combination of controling 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 [1] 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.

[1]For implementations in distributed memory environments, Time Warp with artificial rollback cannot guarantee a space complexity of $O(M^{seq})$. *Cancelback* and Artificial Rollback in achieving the sequential DES storage complexity bound rely on the availability of a global, shared pool of (free) memory.

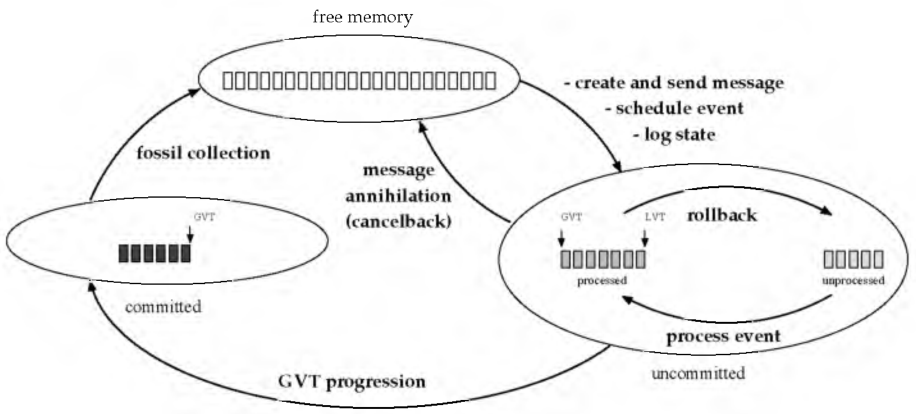


Figure 14: "Flow of buffers" in the AMM protocol

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 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^{uc} 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 LVT will move buffers to M^{uc}, rollback of LVT 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) α, the amount of processed but uncommitted buffers left behind after cancelback, as a parameter to control optimism; and (ii) β, the amount fo buffers freed by Cancelback, as a parameter to control the cycle length. Obviously, α 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). β 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 α and β, but respecting the constraints above. It assumes Cancelback (and fossil collection) overhead to be directly proportional to the Cancelback invocation frequency. Let $\varrho_{M^{uc}} = \frac{e_{committed}N}{T_{process}} - \varrho_{FC}$ be the rate of growth of M^{uc}, where $e_{committed}$ is the fraction of processed events also committed during the last cycle, $T_{process}$ is the average (real) time to process an event and ϱ_{FC} is the rate of depletion of M^{uc} due to fossil collection. (Estimates for the right hand side are generated from monitoring the simulation execution.) β is then approximated by

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

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, α is a parameter to control the upper tolerable bound for the progression of LVT. To set α 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 $\#(e_{cp})$ 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[2]. Starting now with a high parameter value for α (which will give an observation $\#(e_{cp}) \simeq 0$), α is continuously reduced as long as $\#(e_{cp})$ remains negligible. Rollback thrashing is explicitly tackled by a third mechanism that monitors $e_{committed}$ and reduces α and β to their halves when the decrease of $e_{committed}$ hits a predefined threshold.

Experiments with the AMM protcol 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]The *Critical Path* of a DES is computed in terms of the (*real*) processing time on a certain target architecture respecting *lcc*. 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 [33], which can surpass the critical path. The resulting possibility of so called *supercritical speedup*, and as a consequence its nonsuitability as an *absolute* lower bound reference, however, has made critical path less attractive.

3.2.8 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_i, e.g. for fossil collection (*S3.11*) or in the simulation stopping criterion (*S3*). The definition of $\mathrm{GVT}(T)$ has been given in Section 3.2.6. An essential property of $\mathrm{GVT}(T)$ 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 $\mathrm{GVT}(T)$ (*S3.10*) generally is hard, such that in practice only estimates $\widehat{\mathrm{GVT}}(T) \geq \mathrm{GVT}(T)$ are attempted. Estimates $\widehat{\mathrm{GVT}}(T)$, however, (as a necessity to be practically useful) are guaranteed to not overestimate the *actual* $\mathrm{GVT}(T)$ and to eventually improve past estimates.

GVT Computations Employing a Central GVT Manager Basically $\widehat{\mathrm{GVT}}(T)$ can be computed by a central GVT manager broadcasting a request to all LPs for their current LVT and while collecting those values perform a *min*-reduction. Clearly, the two main problems are that (i) messages in transit potentially rolling back a reported LVT are not taken into consideration, and (ii) all reported $\mathrm{LVT}_i(T_i)$ 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 $[T_i^>, T_i^<]$ for every LP_i such that $T_i^> \leq T_i = T^* \leq T_i^<$ for all LP_i. T^*, thus is an instant of real time that happens to lie within every LP's interval.

Samadi's algorithm [56] 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, $\mathrm{LGVT}_i(T_i)$) to the GVT master. An improvement of Samadi's algorithm by **Lin and Lazowska** [39] does not acknowledge every single message. Instead, to every message a sequence number is piggybacked, such that LP_i can identify missing messages as gaps in the arriving sequence numbers. Upon receipt of a control message, the protocol sends out to (all) LP_j the smallest sequence number still demanded from LP_j as an implicit acknowledgement of all the previous messages with a smaller sequence number. LP_j receiving smallest sequence numbers from other LPs can determine the messages still in transit and compute a lower bound on their timestamps.

To reduce communication complexity, **Bellenot's algorithm** [7] 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 [7]). The algorithm efficiently utilizes the static MRG topology and operates in three steps:

(1) (**MRG forward phase**) LP_0 (GVT manager) sends a `GVT-start` to the (one or) two successor LPs on the MRG. Once an LP_i has received `GVT-start`s from each successors, it sends a `GVT-start` in the way as LP_0 did. Every `GVT-start` in this phase defines $T_i^{>}$ for the traversed LP_i.

(2) (**MRG backward phase**) The arrival of `GVT-start` messages at the last node in MRG (LP_N) defines $T_N^{<} = T^*$. Now, starting from LP_N, `GVT-lvt` messages are propagated to LP_0 traversing MRG in the opposite direction; $T_i^{<}$ is defined for every LP_i. Note that LP_i propagates "back" as an estimate the minimum of LVT_i and the estimates received. When LP_0 receives `GVT-lvt`s from its child LPs in the MRG, it can, with LVT_0, determine the new estimate $\widehat{\mathrm{GVT}}(T^*)$ as the minimum over all received estimates and LVT_0.

(3) (**broadcast GVT phase**) $\widehat{\mathrm{GVT}}(T^*)$ 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 (**pGVT**) algorithm [21] 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 (LGVT) should be reported to the manager. The algorithms executed by the GVT manager and the respective LP$_i$'s are described as follows:

GVT mamanger

(1) Upon receipt of LGVT$_i$ determine new estimate $\widehat{\text{GVT}}'$. If $\widehat{\text{GVT}}' > \widehat{\text{GVT}}$ then

(2) recompute the k-sample average GVT increment as

$$\overline{\Delta_{\text{GVT}}} = \frac{1}{k} \sum_{j=n-k}^{n} \Delta_{\text{GVT}_j}$$

where Δ_{GVT_j} is the j-th GVT increment out of a history of k observations, and

(3) broadcast the tuple $\langle \widehat{\text{GVT}}', \overline{\Delta_{\text{GVT}}} \rangle$ to all LP$_i$.

LP$_i$, independently of all LP$_l$, $i \neq l$

(1) recalculate the local GVT estimate

$$\text{LGVT} = \min(LVT_i, ts(m_j) \in \text{OQ}_j)$$

where $ts(m_j)$ is an *unacknowledged* output message, and

(2) estimate K, the number of Δ_{GVT} cycles the reporting should be delayed, as the *real* time $t_{\text{s+ack}}$ necessary to send a message to and have acknowledged it from the manager divided by the k-sample average *real* time in between two consecutive tuple arrivals from the manager as

$$K = \frac{t_{\text{s+ack}}}{\frac{1}{k}\sum_{j=n-k}^{n} \Delta_{\text{RT}_j}}$$

and

(3) send the new LGVT information to the GVT manager whenever $\widehat{\text{GVT}} + K\ \overline{\Delta_{\text{GVT}}}$ exceeds the local GVT estimate LGVT$_i$.

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 $ts(m) \leq LGVT$ 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 $\widehat{\text{GVT}}$), 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 [11] find a straightforward application, solutions more efficient than message ackowledging, the delaying of sending event messages while awaiting control messages or piggybacking control information onto event messages are desired. Mattern [45] uses a

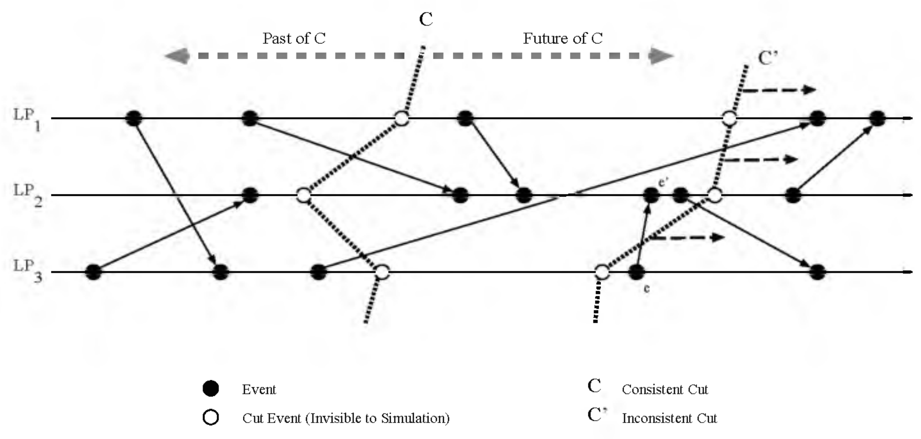


Figure 15: Matterns GVT Approximation Using 2 Contiguous Cuts C and C'

"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 '$\rightharpoonup$' Lamport's *happens before* relation [36] 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 $e \rightharpoonup e'$,

(2) if $e \in SE$ and $e' \in RE$ is the corresponding receive event, the $e \rightharpoonup e'$

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

A *consistent cut* is now defined as $C \subseteq E$ such that

$$(e' \in C) \wedge (e \rightharpoonup e') \Longrightarrow (e \in C).$$

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 $e' \in C'$, $e \rightharpoonup e'$ but $e \notin C'$ (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$_i$, i.e. all the event occurrences recorded in IQ, OQ, and SS up until the cut line, and the *state of the channels* $ch_{i,j}, (i \neq j)$ 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$_{\text{init}}$ initiates the snapshot algorithm by broadcasting *red* control messages to all other LP$_j$ $(i \neq j)$. LP$_{\text{init}}$ 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$_{\text{init}}$. (*White* messages received by a *red* LP are exactly the ones considered as "in transit".)

(4) After LP$_{\text{init}}$ has received all $\Sigma_i(C')$ (including the respective LVT$_i$'s) and the last copy of all "in transit" messages, it can determine C' (i.e. the union of all $\Sigma_i(C')$). (Determination of when the last copy of "in transit" messages has been received itself requires the use of *distributed termination algorithm.*)

Note that the notion of a *local snapshot* $\Sigma_i(C')$ here is related to the cut C', as opposed to its relation to real time in Section 3.2.6. All $\Sigma_i(C')$'s are drawn at different real times by the LPs, but are all related to the same *cut.* We can therefore also not follow the idea of constructing a global snapshot as $\Sigma(T) = \bigcup_{i=1}^{N} \Sigma_i(T) \cup \mathrm{CS}(T)$ by combining all $\Sigma_i(T)$ and identifying CS(T), wich 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 "in transit" 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 *along* a "future" cut C' given the current cut C, C' has to be computed following the algorithm above. Determining the minimum of all local LVT$_i$'s from the $\Sigma_i(C')$s is trivial. To determinine the minimum timestamp of all the message "in transit"-copies 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 *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 "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 [45].

3.2.9 Limiting the Optimism to Time Buckets

Quite similar to the optimistic time windows approach, the *Breathing Time Bucket* (BTB) protocol addresses the antimessage 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 antimessage by restricting potential rollback to affect only local history records (as in SRADS [20]). BTB basically processes events in *time buckets* of different size as determined by the *event horizon* (Figure 16). Each bucket contains the maximum amount of causally independent events which can be executed concurrently. The *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 "risk free" 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) *asynchronous broadcast* can be employed to exchange *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

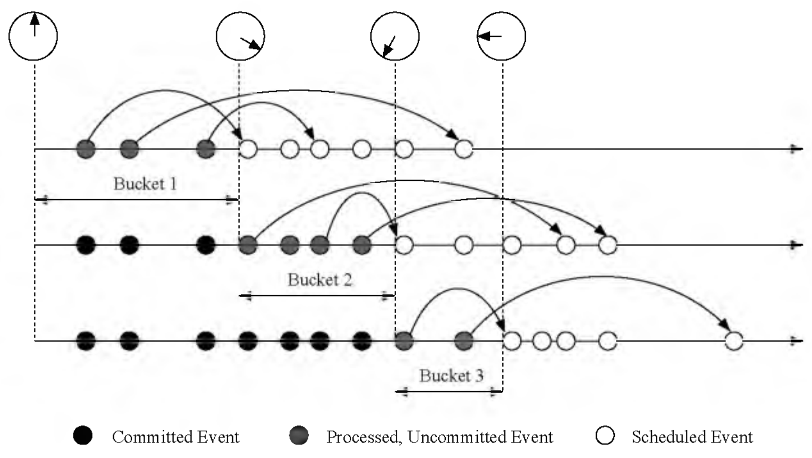


Figure 16: Event Horizons in the Breathing Time Buckets Protocol

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) [60] 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 outputmessages 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 outputmessages 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