Performance Comparable Implementation Design of Synchronization Protocols for Distributed Simulation*

G. Chiola* and A. Ferscha†

* Dipartimento di Informatica, Università di Torino
Corso Svizzera 185, I-10149 Torino, Italy

† Institut für Angewandte Informatik und Informationssysteme, Universität Wien
Lenaugasse 2/8, A-1080 Vienna, Austria

Abstract

A large number of variations for the efficient implementation of distributed simulation protocols on shared as well as distributed memory parallel machines have been proposed in the literature. They are roughly divided into two families: *conservative* protocols (originated by Chandy and Misra) and *optimistic* protocols (originated by Jefferson). Their attained simulation performances as reported in the literature, however, cannot be *compared* directly, due to different implementation strategies, different optimizations and different software environments as well as parallel hardware platforms. Our own implementations of the protocols and some optimized variations on a distributed memory machine (iPSC/860) and on a shared memory machine (Balance B21) revealed the same performance incomparability dilemma. Now we designed a new version of a distributed simulation protocol that has intermediate characteristics with respect to the two main families, and is expected to provide better performance due to a self-adaptive parallelism extraction mechanism. To rank the protocols accurately enough with respect to their execution performance, a performance prediction based on models and behavioral assumptions is just not feasible, so that the only practical evaluation consists in implementing different alternatives and measure their relative performance on some significant test cases.

With this paper we address an evident problem in distributed simulation: the implementation of *different* synchronization protocols, on *different* multiprocessor hardware platforms while preserving *performance comparability*. We demonstrate the possibility of simultaneous implementation of the three substantially different protocols starting from a single program skeleton. A clear separation of the simulation part and the synchronization part of the protocols and reuse of code for more than one variant is the key concept to achieve several goals including: identification of protocol specific performance influence; identification of hardware performance influences; introduction of a comparable amount of optimizations in all versions to allow "fair" performance comparisons; obtain a protocol implementation that is almost independent of the sequential simulation engine run on each processor; and minimization of programming effort.

*This work was conducted while Alois Ferscha was visiting the Dipartimento di Informatica, Università di Torino during August 1993, supported by a grant of the Academic Senate of the University of Vienna.

1 Introduction

Mathematical modelling and computer simulation techniques are widely accepted techniques to predict the behavior of complex or potentially dangerous system before their actual implementation or operation. Although the use of complex simulation models is hard and costly, simulation is often the only practical analysis means for many industrial and scientific problems. Parallel and distributed simulation techniques have in some practical applications, and at least in theory, improved the performance of large systems simulation, but are still far from a general acceptance in science and engineering. The main reasons for this pessimism are the lack of automated tools and simulation languages suited to parallel and distributed simulation, as well as performance limitations inherent in the simulation strategies.

The idea of distributed discrete event simulation (DDES) is conceptually trivial. A large simulation model is (statically) partitioned into a number of smaller submodels. Then the simulation of each submodel is carried out by a dedicated "logical process" (LP). All logical processes are running in parallel on different physical processors, and have to exchange information in order to agree on the current state of the "border" between pairs of "adjacent" submodels. The implementation of DDES requires a sophisticated communication protocol for the exchange of information among logical processes in which each message is attached a *time stamp* that represent the *local virtual time* of the sender logical process at the time the message was produced. In order to preserve causality constraints among events being processed locally in some LP, two main protocols have been proposed for the implementation of DDES: the *conservative* protocol enforces time stamp order processing by restricting the parallelism among LPs, whereas the *optimistic* protocol instead allows out of timestamp order processing, "undoing" that part of the simulation based on a causality fault discovered later in simulated time. Both, the *conservative* [1] and the *optimistic* strategy [15] have led to implementations of general purpose discrete event simulators on parallel and distributed computing facilities (for a survey see [16], [10]).

In conservative DDES strategies all interactions among LPs are based on timestamped event messages sent between the corresponding LPs. Out-of-time-order processing of events in LPs is prevented by forcing LPs to block as long as there is the possibility for receiving messages with lower timestamp, which in turn has a severe impact on simulation speed. Optimistic DDES strategies weaken this strict synchronization rule by allowing a preliminary causality violation, which is corrected by a (costly) detection and recovery mechanism usually called rollback (in simulated virtual time). In [21] a DDES communication protocol (deadlock avoidance) is proven to be correct up to the time revealed by the smallest message timestamp in the system, and free of deadlocks caused by cyclic waiting for messages. The protocol introduces null messages to announce the absence of messages and promise that no message will be sent with a timestamp less than that of the null message. Especially in DDES this approach encounters a dramatic burden on the simulation [8], and additionally does not cope with cycles of zero timestamp increment (a severe source of deadlock). A radical elimination of the use of null messages [2] (deadlock detection and recovery) allows the occurrence of deadlocks, and proposes a global controller process to detect deadlocks, which is broken by allowing the LP with the smallest timestamp to proceed (which is always safe). Naturally a global controller violates the principles of distributed computing. Current activities with the conservative approach concentrate on the detection of optimal lookahead information to speed up simulation [18, 25, 23].

In order to guarantee a proper synchronization among LPs in optimistic simulation, two rollback mechanisms have been studied. In the original Time Warp [15] [14] an LP receiving a message with timestamp smaller than the LPs locally simulated time (*straggler message*) starts a *rollback* procedure right away. Rollback is initiated by revoking all effects of messages that have been processed prematurely by restoring to the nearest safe state, and by neutralizing, i.e. sending "antimessages" for messages sent since that state (*aggressive cancellation*, *ac*). The impact of the erroneously sent messages is that also succeeding LPs might be forced to rollback, thus generating a rollback chain that eventually terminates. Reducing the size of the rollback chain is attempted by "filtering" messages with preempted effects [24], by postponing erroneous message annihilation until it turns out that they are not reproduced in the repeated simulation (*lazy cancellation*, *lc*) [11], or by maintaining a causality record for each event to support *direct cancellation* of messages that will definitely not be reproduced [17]. The performance of rollback mechanisms is investigated in [19] and [20].

Eventually we also came up with our own proposal for a new DDES protocol that we called *probabilistic* (as opposed to the optimistic and conservative ones). The motivations and the description of the probabilistic DDES protocol are reported in [7]. In principle, our probabilistic protocol is a performance efficient compromise between the two classical approaches: consider that some external event ee_i occurs in some time interval $[s, t]$, and let the probability of the actual occurrence time $occ(ee_i) = t*$ be equally likely for all $s \leq t* \leq t$. Then the conservative simulation protocol could have produced real simulation work by progressing over s with probability $(1 - \alpha)$ during the time interval $[s, s + \alpha(t - s)]$. So at least for small values of α the conservative protocol seems overly pessimistic. The optimistic protocol on the other hand will proceed simulating internal events even after surpassing t. This is overly optimistic in the sense that every internal event with $occ(ie_j) > t$ will definitely have to be invalidated. The probabilistic protocol weakens the conservative "block until safe-to-process"-rule in a sense that it allows progressing simulation up until $\widehat{t}(O)$, $s \leq \widehat{t}(O) \leq t$, where $\widehat{t}(O)$ is an estimate based on the arrival instants $O = (o_1, o_2, \ldots o_n)$ observed during a time window of length n by some LP. Compared to the optimistic strategy it prevents from propagating incorrect computations too far ahead into the simulated future, and thus avoids unnecessary communication overhead by breaking rollback cascades as early as possible. Moreover, the arrival patterns observed in O are used to *adapt* the logical process to a synchronization behavior that is the best tradeoff among blocking and optimistically progressing with respect to the parallelism inherent to the simulation model.

Performance comparison of DDES is usually based on *critical path analysis* [19], which is performed upon an event precedence graph (EPG) representation of the causality dependencies among events, with (real) message propagation delays associated with the message sending arcs of EPG, and (real) event execution times associated with the event nodes. The length of a critical path of EPG is known to be a lower bound on the real simulation time using *any* conservative simulation mechanism. In [13] it is stated (and shown) that any simulation mechanism able to outperform this lower bound *must* be optimistic, and examples are constructed in which Time Warp with *lc*, Time Warp with *lazy rollback* (*lr*) and Time Warp with *phase decomposition* (*pd*) gain so called "supercritical speedup" (beat the lower bound), whereas Time Warp with *ac* cannot (*lr* means that re-execution caused by some rollback is invoked only for erroneous states and at the latest possible time). Of course, critical path analysis cannot help with the problem of comparing *implementations* of the protocols in specific environments, since it does not account for hardware performance and performance influences imposed by the environment as such. In order for that, system models explicitly representing *all* the performance influencing factors of the target system as well as an explicit representation of the simulation

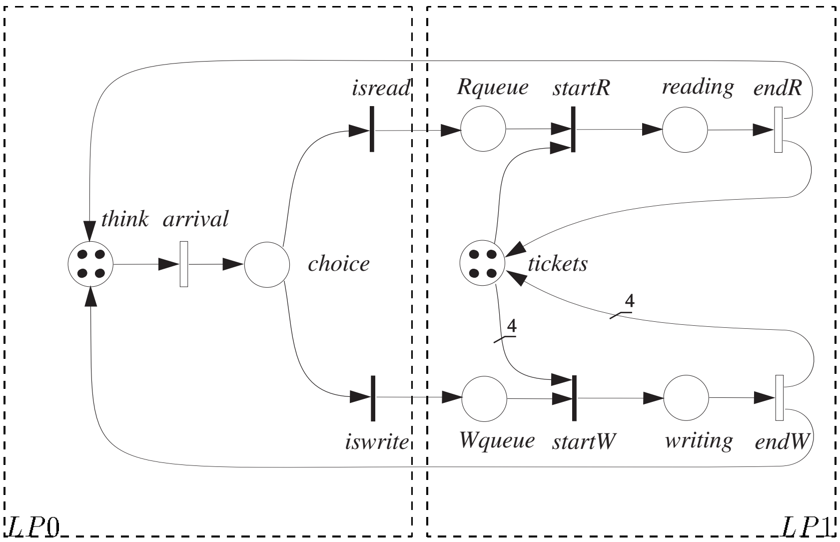

Figure 1: A GSPN Model of a Multiple Reader Single Writer System.

model would be necessary to judge upon the performance of the different strategies. Although models of that kind have been developed based on plausible assumptions and hypothetical workloads (e.g. PHOLD) [9] [12] [22], the only practical chance of a "convincing" comparison is the one based on real implementations. Although a "*fair*" comparison of the protocols is (obviously) highly demanded by the distributed simulation community, no serious attempts for it have been reported in the literature.

The paper is organized as follows: First we give a brief motivation for a performance comparable implementation of the three principle protocols by demonstrating phenomena that we observed with our previous implementations. We work out a design of the implementation in Section 2, and provide a sketch of the protocols in Section 3. A detailed implementation skeleton for the simultaneous realization of the three protocols is given in Section 4. The skeleton can be used directly to produce one of the specific protocols by code extraction using the optional compilation facility of the `cpp` language processor.

1.1 Performance Incomparability

In previous work [5] we provided a DDES implementation for generalized stochastic Petri net (GSPN) models [3], where the idea was to exploit Petri net analysis techniques to identify potential model parallelism and come up with a good static partitioning of the model based on this information. We identified several criteria to automatically extract lookahead from a model and to improve the efficiency of rollback mechanisms for some particular model structures. However the choice of the basic DDES protocol was not obvious, and we just tryed out different prototypes to find out which one was performing better on some test cases [6, 4]. From the measurement of these prototypes we gained insights on the impact of the different parameters of the hardware and the protocol organization on the performance, yet we could not conclude whether conservative protocols would perform better than optimistic ones or vice-versa, due to the strong effect of some "minor optimizations" (lazy rollback, message send ahead, etc.) introduced in some of the prototypes and not in others [5].

To explain this performance incomparability look at Figure 1, which gives a GSPN (Generalized Stochastic Petri Net) model of the "multiple readers single writer" access to a shared resource. The

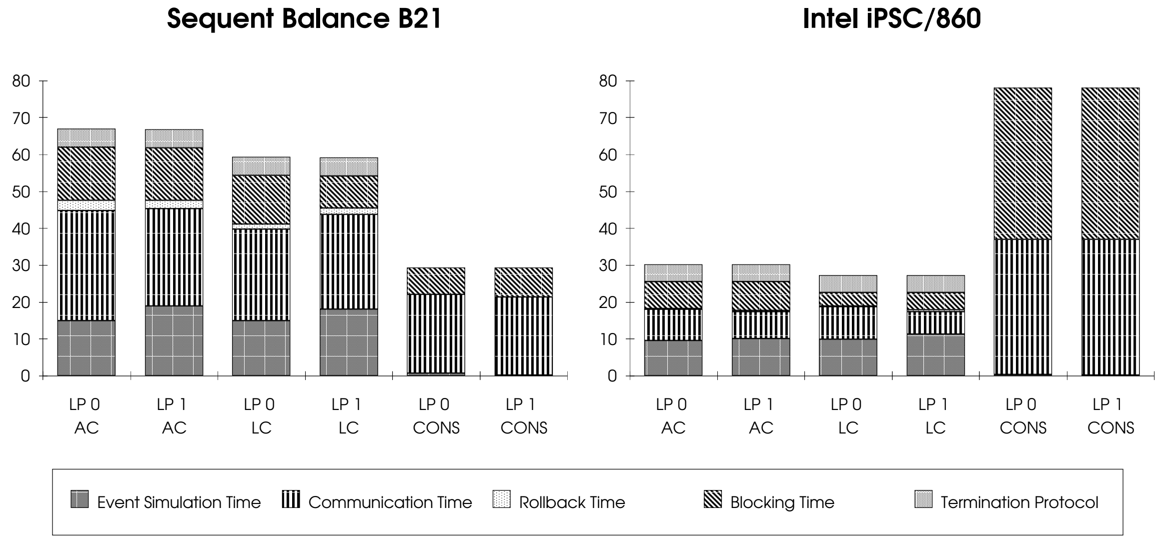

Figure 2: Performance of Optimistic and Conservative DDES Protocols

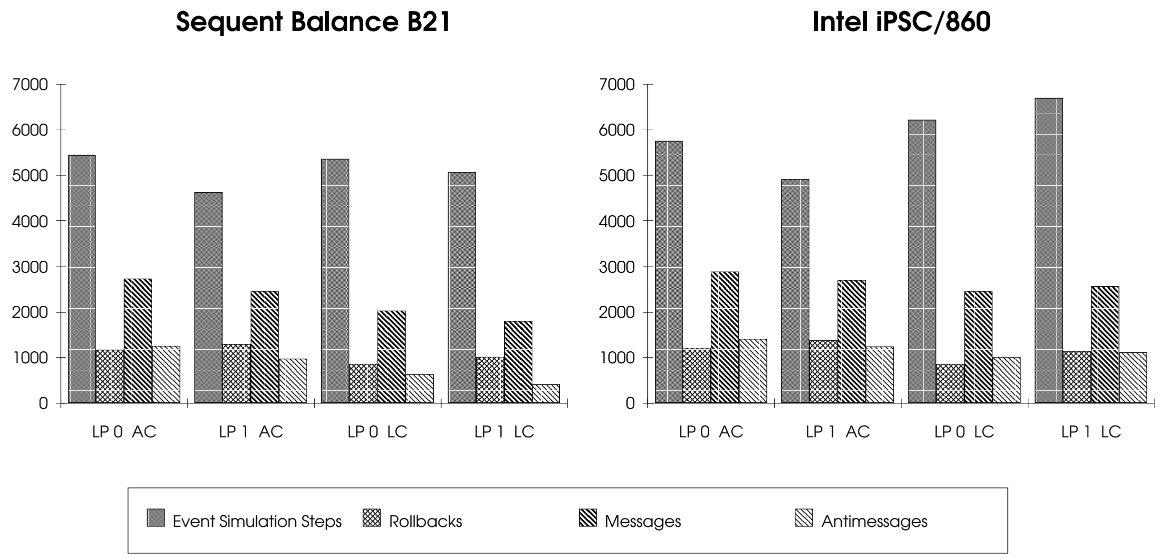

Figure 3: Event Counts of Optimistic and Conservative DDES Protocols

GSPN comprises four transitions having a *firing delay* of $\tau = 0$, and three of them having a $\tau \sim exp(\lambda)$ which is an exponentially distributed random variate with parameter λ: the former are responsible for modelling the behavioral structure of the system, the latter describe the timing aspect. In the particular example, $N = 4$ processes share the common resource. The arrival of processes is modeled as an exponentially timed transition *arrival* in the following way: whenever a token (process) happens to arrive in place *think*, an exponentially distributed random variable is drawn, and its value is taken as the firing time of transition *arrival* for this token. Note that *arrival* can happen simultaneously for several tokens presently in *think*. Once *arrival* fires, the corresponding token (process) is deposited in the place *choice*, enabling *isread* and *iswrite* simultaneously. A so called random switch defined upon *isread* and *iswrite* in terms of selection probabilities is used to decide which of the two conflicting transitions should actually fire. Given that *isread* fires, then the token (process) is "queued" into

Rqueue. Should there be at least one token in place *tickets*, denoting the availability of "entry tickets" for accessing the shared resource, then *startR* fires (in zero time), and the process starts a reading access (token in *reading*). Obviously several read accesses can be performed in parallel, with respect to the availability of "tickets" (tokens) in place *tickets*. After the completion of the read access – which again takes an exponentially distributed amount of time per process – the ticket is released again into *tickets*, and the process returns to become a new arrival. Should instead *iswrite* be fired with the a token in *choice*, then the corresponding process would be "queued" in *Wqueue*. For firing *startW* $N = 4$ (all) tickets are required to be available in *tickets*, expressing the fact that a write access precludes simultaneous reads.

Simulation of the sample net after a partitioning into two logical processes LP0 and LP1 (firings of transitions *arrival*, *isread* and *iswrite* are simulated by LP0, and firings of transitions *startR*, *startW*, *endR* and *endW* are simulated by LP1) on two conceptually different multiprocessor systems gave the performance results depicted in figure 2. For the particular example GSPN we have selected a "balanced" parametrization ($\lambda_{arrival} = 1.0$, $\lambda_{endR} = 2.0$, $\lambda_{endW} = 0.5$, $prob(isread) = 0.8$, $prob(iswrite) = 0.2$, i.e. the write access takes (on average) four times as long as the read access, whereas the chance of an arbitrary process requesting read access is 0.8, and the expected arrival delay is 1.0). The expected number of events (transition firings) in LP1 per firing of *arrival* is 2, thus 2 events in LP0 cause 2 events in LP1 on average (causality constraints of transition firings neglected). The "absolute" performance figures in Figure 2 show measurements of different protocol implementations on the Intel iPSC/860 hypercube and the Sequent B21; AC (LC) is the optimistic protocol with aggressive (lazy) cancellation, while CONS is the conservative protocol implementation. The model was simulated up to *local virtual time* LVT=1000 (For presentational reasons, CONS on the B21 represents simulation up until LVT=100 (!)); Figure 2 (absolute execution times) and Figure 3 (event counts) show average values of 30 independent simulation runs.

Hardware Performance At a first glance we note that communication latency is dominating the processing power for the particular application on both machines, independent of the DDES protocol. Higher load in the LPs, i.e. a larger set of transitions with local enabling effects would improve the communication/computation ratio, which is necessary to achieve real speedup over the sequential simulation [5], but the communication dominance in DDES implementations on contemporary parallel platforms is evident. Another obvious performance observation is, that the underlying hardware has a serious impact on the DDES behavior and performance: For the case of the B21 (shared memory machine) we would expect to see that the (maximum) channel bandwidth of the system bus of 80 MByte/s (8 Byte, 10 MHz) allows higher (synchronization-) message throughput than the (maximum) channel bandwidth of 2.8 MByte/s of the communication network of the iPSC (distributed memory machine) does. In fact, since the current implementations of the protocols use 10 Byte messages, communication latency between two adjacent nodes on the iPSC/860 using synchronous message passing ($T_{comm}(B) = 69\mu\text{sec} + B * 0.43\mu\text{sec}$, B = number of Bytes)) is faster than on the B21 in absolute terms. This can be seen from the absolute communication times if Figure 2 consumed for communicating (approx.) the same amount of messages (Figure 3). We can also see that the iPSC's i860 processor (80 MFLOPS peak) outperforms the B21's NS32032 (2.25 MFLOPS peak) also in event processing by several times.

Protocol Performance A curious phenomenon is observed when comparing the absolute performance of the conservative with the optimistic protocols (Note again, that CONS on the Balance in Figure 2 shows only 1/10 of the real computation time): both AC and LC outperform CONS - on both machines -, but CONS does not seem to do any event processing. In fact CONS *does* simulate all the transition firings in the simulation model, but does not have to log states for occasional rollback. We can see that almost all the *event simulation time* in AC and LC is used for memory access to store states; the computational requirements for 'firing one transition' is almost neglectible (a couple of processor cycles). CONS is heavily overloaded with sending (mainly) null messages, respectively blocked when awaiting a message. LC improves absolute performance of AC by breaking cascades of rollback, mainly in LP0. The reduction of the amout of antimessages originating from rollbacks thus cuts unnecessary communication overhead and speeds up execution time (in our example by about 10 %).

"Interwined" Performance An example where hardware performance is interwined with protocol performance is seen when comparing the absolute number of events processed in AC and LC on the different machines as depicted in Figure 3. Looking at the iPSC part one may realize, that LC increases the absolute number of event simulation steps slightly and one would expect the overall execution time (Figure 2) to increase, but the contrary is the case. As supposed, LC reduces the amount of rollbacks, and as a consequence, the number of antimessages; since communication is far more 'expensive' than event processing, the communication overhaed reduction overwhelms the increased computations. This argument, although not explicit in Figure 3, is consistent with the B21. Since simulation performs 'slower' (in processor time) on the B21, LC finds higher chances for local antimessage annihilation and can break relatively more rollback chains, hence decreases the amount of antimessages to a higher extent than the 'faster' iPSC. Curiously, in this case, 'slower' is better for speeding up the simulation.

To conclude the motivation for our work, we see that measurements and execution traces provided insight in performance influence phenomena that would not have been recognized by analytical DDES models, not even by simulation models because of the high complexity of the protocols and the intervention of hardware performance, which is extremely hard to quantify. Hence, for a fair ranking of DDES protocols, we have to turn to an actual implementation of an 'equally' optimized program for each of the protocols to be run on a state-of-the-art technology parallel machine.

2 A Performance Comparable Implementation Design

Any simulation program is composed of a *static* simulation kernel (SK) which implements a simulation environment, and a *variable* part describing the simulation model (SM) targeted to the environment. In this sense the SK represents an abstract machine, interpreting the arbitrary simulation application by managing event scheduling and event processing as defined by the variable model description. A distributed simulation program now implements a *distributed* SK in terms of a set of communicating LPs, executing on a parallel or distributed host system. With the concept of LPs a partitioning of the simulation model data (rather than a functional partitioning) is supported in order to divide the computation among a set of physical processors (see the spatial decomposition of the GSPN in Figure 1 as an example).

For the purpose of a performance comparable implementation design, we look at a *logical process*

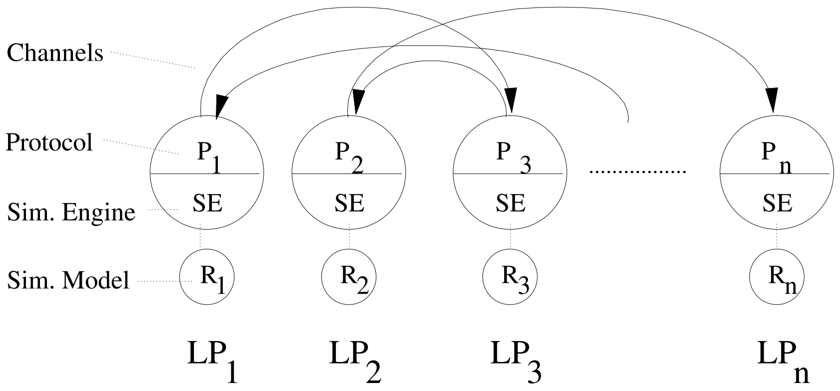

Figure 4: Logical Process Implementation of a Distributed Simulation Program.

(LP) as a tuple $LP_k = (R_k, P_k, SE)$, where R_k is some part of the SM data (in the case of GSPN simulation models a spatial *"region"*, i.e. a subset of the places, transitions and arcs of the GSPN), P_k is the communication and synchronization protocol of LP_k and SE is a simulation engine executing SK functions in the domain of R_k. The set of all LPs $LP = \bigcup_k LP_k$ together with directed communication channels $CH = \bigcup_{k,i} ch_{k,i} = (LP_k, LP_i)$ constitute the *Graph of Logical Processes* $GLP = (LP, CH)$. Figure 4 depicts a GLP of a distributed simulation program.

The distinction of the LP components R_k, P_k and SE is motivated by the goal of protocol performance comparison. Although partitioning SM into R_k's is the most serious issue with respect to the attainable speedup of a distributed simulation program [6, 5] we isolate it here from P_k in the same way as we do with SE. SE is supposed to perform the simulation of event occurrences, scheduling, virtual time management etc., and is specific to the simulation domain (e.g. queueing network models, Petri nets, etc.). With P_k we address LP_k's interface to other LPs, i.e. the way how the distribution and collection of incoming and outgoing messages are handled, the DDES strategy applied (in further C for conservative, O for optimistic and P for probabilistic), and how the SE is driven. Implementing P_k is of main interest for the rest of this work, since performance influences stemming from R_k or SE can easily be identified in a ceteris paribus analysis, and are independent of the P_k performance.

2.1 Implementing a CPO Superposition

With careful comparison of the three protocols C, P and O we identified a considerable amount of code that could be shared by the three protocols: both the conservative and the optimistic protocols could be programmed using subsets of functions already needed by the probabilistic protocol. Also the data structures used for the probabilistic protocol can mainly be realized by a combination of the data structures, thus allowing a separation of data needed by the other protocols. This separation is clearly worked out in Figures 6 and 7, and is explained in detail in the next section. The separation at the level of program code is also presented there.

The performance *comparability* of the different protocol implementations is now achieved as follows (see Figure 5). From the core code and data structures shared among the three protocol implementations, one of the protocols C, P or O is automatically extracted, and the proportion of "shared" to "protocol specific" amount of code/data structures is recorded by annotating lines of code that would have been extracted for every protocol. After compilation of the extracted protocol it can be executed

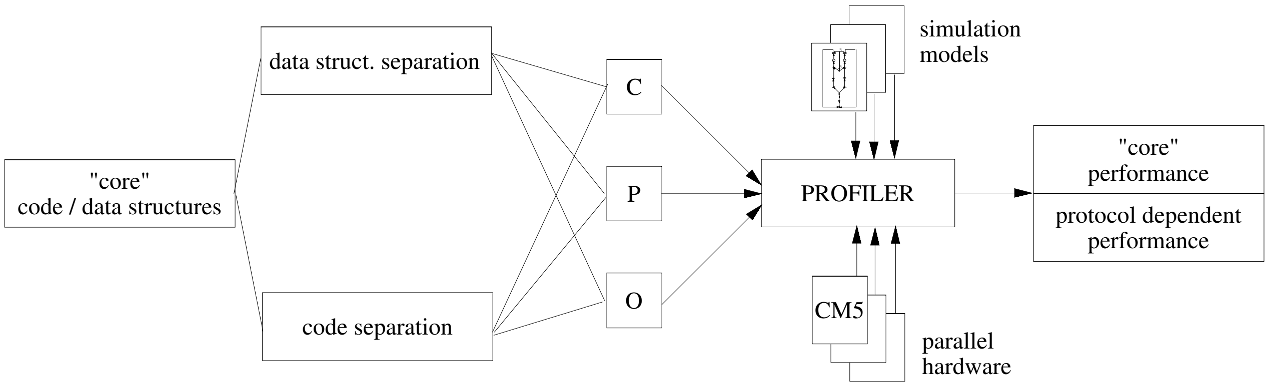

Figure 5: Preserving Performance Comparability of the Protocols.

to simulate a specific simulation model on a given hardware platform. During this execution a program profiler collects trace information on local memory access, processor and communication activity, as well as on the level of the simulation protocol. The execution trace distinguishes operations according to their annotation, such that the performance (figures) derived from the traces can be interpreted with respect to their cause, namely as independent of the protocol or protocol dependent. A *direct* comparison of the DDES performance is always valid for the shared code parts, while a detailed analysis is necessary for a judgement on the protocol related performance as such. In other words, the ranking of the protocols can be based on the protocol specific performance, and obviously we can also quantify the effects of protocol specific optimizations, since they will always show up in the protocol dependent performance. As an example, if we wanted to compare the performance of the O and the P protocol (ceteris paribus) we would identify the data structures in the (shaded) area *cpo* and *po* of Figure 6 and 7 together with the annotated operations in the pseudocode as the core of both protocols. Then trace data collected by profiling operations manipulating data in *p* (which is neither in *cpo* nor in *po*) would be used to argue on the "performance gap" among O and P. If we wanted to compare P and C, we would argue on profile information originating outside the greatest common part of shared code, namely *cpo*.

One can see, that this strategy will also provide us with the possibility of porting the codes to other high performance computing environments and allow a comparison of the performance on different hardware platforms in absolute terms (i.e. comparison against each other as well as against the best sequential simulation engine available).

3 Sketch of the Protocols and their Data Structures

The implementation of a DDES strategy consists of a decomposition of the "sequential" discrete event simulation task into a set of communicating LPs to be assigned to individual processing elements working autonomously and communicating with each other for synchronization. Each and only one LP is assigned to a dedicated processor and is residing there for the whole simulation elapsed time. A message transportation system is assumed to be available (either implemented fully in software or by hardware routing facilities) that implements directed, reliable, buffered FIFO communication channels among LPs. Every LP applies one and the same discrete event strategy to the simulation of the local submodel. We are not interested here in the sequential simulation part of the LPs but only in their communication protocol (which includes the management of the message time stamps).

The general structure of an LP implementation can be sketched as an initialization in which the local simulation state is started from virtual time zero and the initial state of the model, followed by a main loop that is repeated until the end of the simulation run. The main loop comprises the following steps:

L1 Look in the input buffer for possible incoming messages; determine their time stamp in order to establish their relative occurrence time order with respect to the locally simulated events.

L2 Send out the messages that are possibly stored in the output buffer as a result of previous local simulation steps; add to these messages informations that are needed by the protocol in order to properly handle the time stamp information and the local virtual time progression.

L3 Depending on the protocol, determine a *local virtual time window* (LVTW) up to which the LP is allowed to simulate local events.

L4 For each event with scheduled time less than or equal to LVTW simulate the event in time stamp increasing order; the simulation of each event may involve the generation of output messages directed to other LPs that are stored in the output buffers to be actually sent in step L2 of the next iteration of the loop.

The case of conservative protocols is conceptually simpler. In this case the protocol selects the maximum value for LVTW in step L3 such that time stamp increasing ordering of received messages is guaranteed. The simulation steps performed locally in step L4 are thus guaranteed to be consistent with the states of the other LPs. In order to avoid deadlocks due to circular waiting, it is necessary for the pessimistic protocol to send a special type of messages called *null messages* in step L2 even if there is no real message to send. Null messages diffuse the best lookahead information available locally on a lower bound for the time stamp of the next message that will be sent in future steps. A significant lookahead information may allow the receiver LP to increase its LVTW value so as to break the circular waiting deadlock.

In the case of optimistic and probabilistic protocols, increasing time stamp ordering is not guaranteed by the communication protocol, so that an LP executing step L1 may find an input message with a time stamp lower than the current value of the LVT (*straggler message*). This implies an inconsistency of the current state of the LP that received the straggler message, since it already performed some simulation steps without properly taking into account the effect of the straggler message on the local simulation state. This inconsistency is removed by a *rollback*, i.e., restoring the last state implying a local virtual time less than or equal to the time stamp of the straggler message, and redoing the simulation starting from that point. The implementation of the rollback feature requires storing the complete local state description for each past simulation state, so that the state information can be retrieved in case a straggler message is received with time stamp greater than or equal to the LVT of that state. An efficient implementation from the point of view of memory usage requires the computation of the *global virtual time* (GVT) (or at least a reasonable lower bound estimate for it) defined as the greatest LVT value for which an LP is guaranteed not to receive a straggler message. In our implementation of the optimistic and probabilistic protocols we explicitly included an estimate of

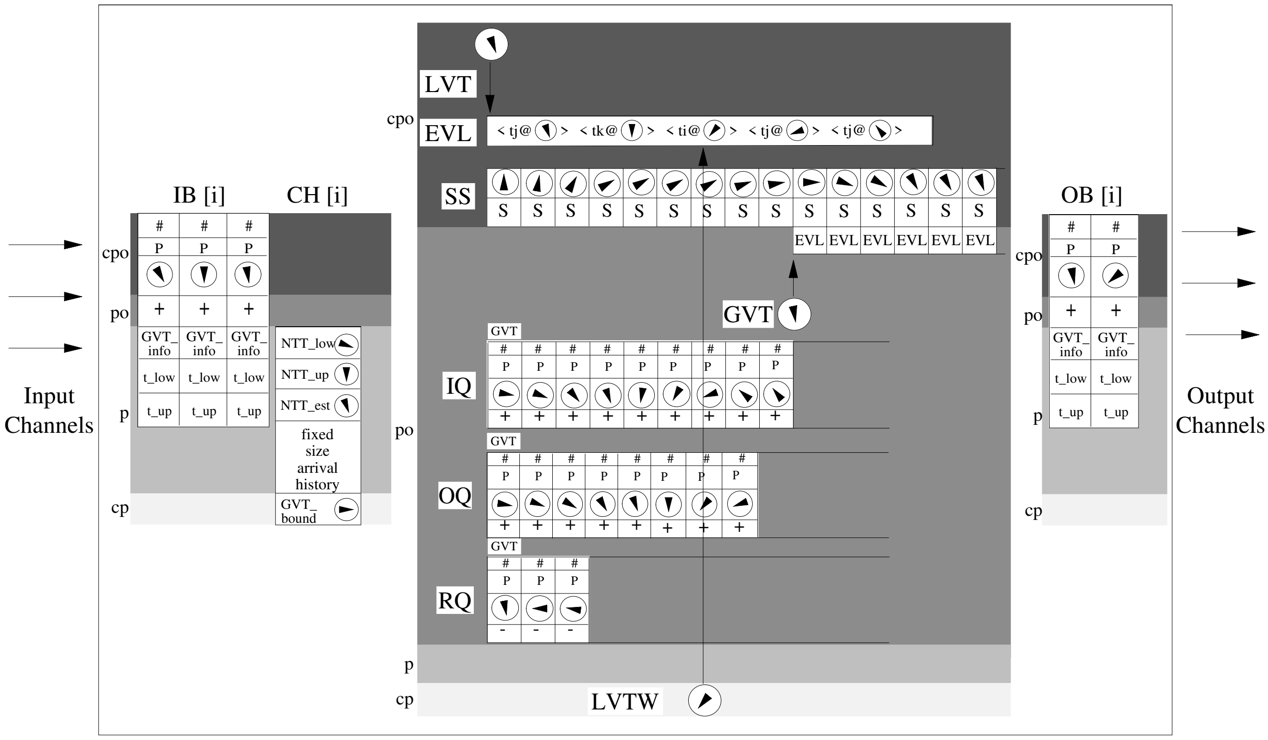

Figure 6: Pictorial Representation of the Common Data Structure.

a lower bound for the GVT of each LP that is used to purge the record of rollback related information in order to reduce the memory requirements. The estimate of the GVT is based on the diffusion of additional time stamp informations attached to outgoing messages in step L2. The estimate itself is computed on a local basis using essentially the same time stamp algorithm as for the computation of the LTVW in the conservative protocol.

The rollback may also imply the "undoing" of some communication with other LPs. If the messages to be revoked are still in the output buffer they can be just deleted in step L1, thus avoiding their sending in step L2. Otherwise, an *antimessage* has to be sent to force the receiver LP to cancel the erroneously sent information. The standard implementation of the optimistic protocol does not include the notion of a time window, so that the value of LVTW can be just set to infinity (there are also some variations proposed in the literature in which a time window is artificially imposed so as to prevent one LP to go too far away in the future with respect to the others, thus limiting the length of the rollback in case of straggler messages). The probabilistic protocol, instead, uses a statistical characterization of the time stamp of incoming messages to "guess" the time stamp of the next message (this guess is also educated by the lookahead information coming from the sender LP), and the LVTW value is set so that to keep the probability of a straggler message under a predefined threshold (based on the statistic characterization that is locally computed).

A common data structure has been conceived in order to allow the implementation of all three protocols. A sketch of the data structure maintained by a single LP is depicted in Figure 6, details of data templates are given in Figure 7. We annotate the belonging of data structures to some protocol (C, P, or O) by a shaded area, i.e. templates found in the area *cpo* are common to all the three protocols, while e.g. templates in *po* refer to data which is specific to the protocols P and O.

Three variables are managed by the protocols in order to keep track of the current simulation time:

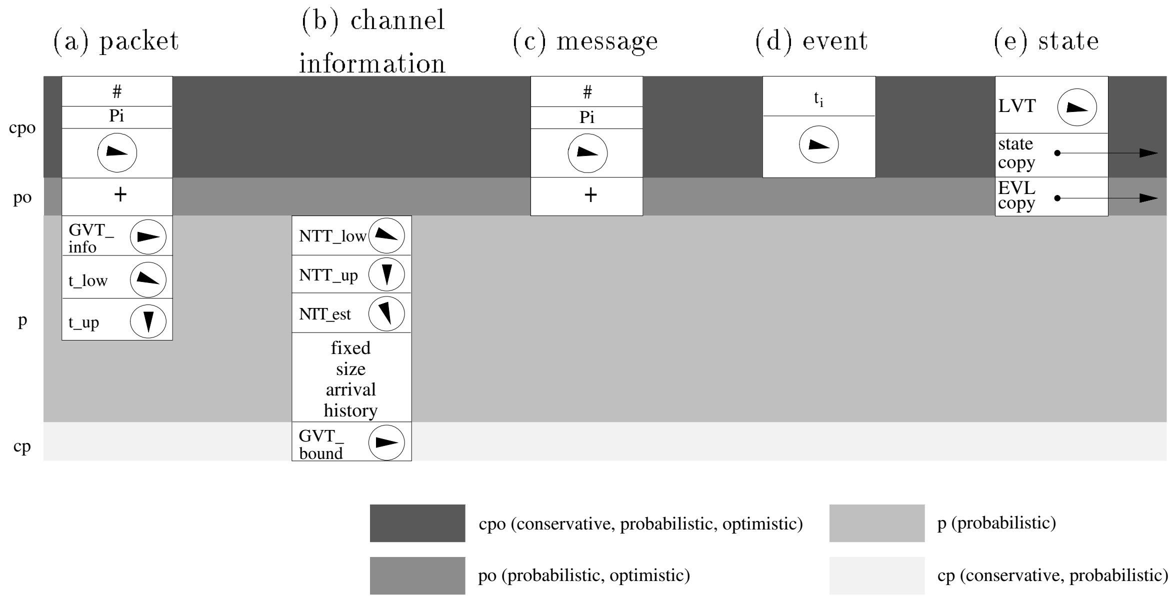

Figure 7: Data Structures.

LVT, LVTW, and GVT. Each input and output communication channel with another LP is associated with a (input or output) buffer that may contain packets (received or going to be transmitted). A packet is composed of a message part, which is the information used in order to carry out the simulation, and additional information needed for the protocol implementation.

The optimistic and probabilistic protocols require the presence of additional structures called the *input* and *output queues* in order to implement the reordering of input messages by increasing time stamp and the sending of antimessages in case of rollbacks. A state stack is also maintained for these two protocols in order to support the rollback mechanism. Information stored in the state stack includes the current state of the local submodel, the current event list, the LVT value, the seed for the pseudorandom number generator used by the simulation engine, etc., so that the LP can actually resume the simulation after a rollback as if the erroneous simulation had not been performed at all.

The implementation of the three protocols has been undertaken using the CMMD message passing library on the CM-5 target parallel machine. All processors involved in the computation are running the same program, but are initialized with data structures corresponding to different simulation submodels. A sequential program implements the behavior of each LP allocated to a processing node of the CM-5. The asynchronous inter processor message passing library is used to implement the buffered send and receive operations required by the three protocols. We will present the organization of the programs disregarding the actual implementation of these low level communication primitives (that are obviously common to the three protocols).

4 Relevant Implementation Details

In the following we describe in a pseudo code fashion the logical process behavior as verbally described in the previous section. The annotation on the left hand side to every statement represents a selection directive for the preprocessor to extract the code required by one of the protocols C, P or O. This annotation is also used for advising the profiler to which trace set to collect the execution information of the statement during a simulation run on the multiprocessor. (By only looking at the left hand side annotation of the code one will realize a surprising amount of overlap in the implementation, which is due to a considerable sophistication we employed in condensing the functionality of the three protocols.)

```
C P O | Statements
------|----------------------------------------------------------
: : :  logical_process(submodel_k) {
: : :   LVT = LVTW = 0.0; /* S1: initialization */
  : :   GVT = 0.0; prot_clear_queues(IQ,OQ,RQ);
: : :   prot_clear_stack(SS);
: : :   prot_init(submodel_k); sim_init(submodel_k);
: : :   for (i=1; i<N_InputChannels; i++){ /* set the initial bounds */
:         Channel[i].Ch_Clock = 0.0;
  : :     Channel[i].GVT_Bound = 0.0;
  :       Channel[i].NTT_low = 0.0; Channel[i].NTT_up = INFINITY;
  :       Channel[i].NTTe = UNDEFINED; Channel[i].CH_history_info = NULL;
: : :    }
: : :   state = create_push_new_state(SS,LVT,sim_getcopy_State()); /* initial state */
  : :   state->Eventlist = sim_getcopy_EVL(); GVTpointer = state;
: : :   LVTpointer = state;
: :     LVT_lookahead = sim_getlookahead(LVTpointer); /* determine initial lookahead */
: : :   do { S2: /* main loop */
:         stop = (LVT > ENDTIME)
  : :     stop = (GVT > ENDTIME);
  : :     straggler = NULL; GVT = INFINITY;
: : :     for (i=1; i<N_InputChannels; i++){ /* S2.1: unload IBs OBs */
: : :       while ( ! prot_empty(IB[i]) ) {
: : :         packet = prot_first_packet(IB[i]);
:           prot_adjust_time(packet->Message->timestamp,&(Channel[i].CH_Clock));
  : :       prot_adjust_time(packet->GVT_info,&(Channel[i].GVT_Bound));
:           if ( packet->type == NULL_MSG ) prot_remove_top(IB[i]);
:            else break;
  : :       GVT = min(GVT, Channel[i].GVT_bound);
  :         if ( packet->t_low != UNDEFINED ) {
  :           Channel[i].NTT_e = UNDEFINED;
  :           prot_adjust_time(packet->t_low,&(Channel[i].NTT_low));
  :           prot_prob_adjust_t_up(packet->t_up,&(Channel[i].NTT_up));
  :          }
  : :       message = packet->Message;
  : :       prot_insert_in_queue(message,packet->type,IQ); /* annihilate if type == ANTI_MSG */
  :         stat_update_arrivalhist(packet,Channel[i].CH_history_info);
  : :       if (message->timestamp < LVT)  /* this is a straggler */
  : :         if ( straggler == NULL || message->timestamp < straggler->timestamp )
  : :           straggler = message;
  : :       prot_remove_top(IB[i]);
: : :      }
: : :    }
:         for (i=1; i<N_OutputChannels; i++) /* fill up empty OBs with nullmessages */
:           if ( prot_empty_buf(OB[i]) ) {
:             packet = newpacket(NULL_MSG, (LVT + LVT_lookahead[i]));
:             prot_insert_in_buffer(packet,OB[i]);
:            }
  : :     if (straggler != NULL) {
```

```
  : :      prot_local_purge_OB(straggler->timestamp,OB);
  : :      LVTpointer = prot_rollback(straggler->timestamp,LVTpointer,OQ,RQ);
  : :      LVT = sim_reset_State_EVL(LVTpointer);
  : :     }
  : :    GVTpointer = prot_remove_EVL_copies(GVTpointer, GVT); /* purge useless EVL copies from SS */
  : :    prot_purge_queue(IQ,GVT); prot_purge_queue(OQ,GVT);
  : :    GVT_lookahead = sim_getlookahead(GVTpointer); /* improve GVT for each output channel */
: : :    for (i=1; i<N_OutputChannels; i++) /* S2.2: send out content of OBs */
  : :      if ( ! prot_empty(OB[i]) {
  : :       /* propagate new GVT estimates by piggybacking to output messages */
  : :        packet = prot_first_packet(OB[i]); packet->GVT_info = GVT + GVT_lookahead[i];
: : :        do {
: : :          packet = prot_first_packet(OB[i]); prot_remove_top(OB[i]);
  : :          prot_insert_in_queue(packet->Message,packet->type,OQ);
: : :          prot_sendout(packet);
: : :         } while ( ! prot_empty_buf(OB[i]) );
  : :       }
: : :    if ( stop ) break; /* stop when reaching ENDTIME after having sent output packets */
: : :    LVTW = INFINITY; /* S2.3: estimate the Local Virtual Time Window*/
: :      for (i=1; i<N_InputChannels; i++) {
  :        if ( Channel[i].NTTe == UNDEFINED ) Channel[i].NTTe = stat_estimate(Channel[i]);
  :        if ( LVTW > Channel[i].NTTe ) LVTW = Channel[i].NTTe;
:          if ( LVTW > Channel[i].CH_Clock ) {
:            LVTW = Channel[i].CH_Clock; first_IB = &(IB[i]);
:           }
: :       }
: : :   S2_4: /* is there something to do within LVTW ? */
: : :    nexteventtime = sim_gettime_first_EVL(); /* returns INFINITY if EVL is empty */
: : :    if ( nexteventtime >= LVTW ) {
:          if ( prot_empty_buf(*first_IB) )
  : :      if ( prot_empty_que(IQ) )
: : :        goto S2;
: : :     }
:        nextmessage = prot_empty(*first_IB) ? NULL : prot_first_packet(*first_IB)->Message;
  : :    nextmessage = prot_get_first(IQ,LVT); /* returns NULL if IQ is empty */
: : :    nextmessagetime = (nextmessage == NULL) ? INFINITY : nextmessage->timestamp;
: : :    outmessagelist = NULL; /* S2.5: simulate one step */
: : :    if (nexteventtime < nextmessagetime) { /* S2.5.1: process first event */
: : :      LVT = nexteventtime; outmessagelist = sim_firefirst_EVL(LVT_pointer);
  : :      for (p=RQ; ((p!=NULL)&&(p->timestamp<=LVT)); p=p->next) { /* Lazy cancellation */
  : :        if (prot_exists_dual_message(outmessagelist, p)) {
  : :          prot_remove_from_list(p,&outmessagelist); prot_remove_from_queue(p, RQ);
  : :         }
  : :         else prot_add_to_list(p,&outmessagelist);
  : :       }
: : :     }
: : :     else { /* S 2.5.2: process next message */
  :         if (nextmessagetime > LVTW) goto S2; /* make sure the message is within LVTW */
:           prot_remove_top(*first_IB);
: : :       LVT = nextmessagetime; sim_use_msg(nextmessage);
: : :      }
: : :    state = create_push_new_state(SS,LVT,sim_getcopy_State()); /* S2.6: record new state in SS */
  : :    state->Eventlist = sim_getcopy_EVL();
```

```
: : :     LVTpointer = state; /* keep it as the new LVTpointer - it will become the top of SS */
: :       LVT_lookahead = sim_getlookahead(LVTpointer); /* ask for lookahead in the new state */
: : :     for (p=outmessagelist; p!=NULL; p=p->next) { /* S2.7: deposit output messages in OBs */
: : :       outbuff = prot_which_LP(p); packet=newpacket(); packet->Message = *p;
  :         packet->t_low = LVT + LVT_lookahead[outbuf]; packet->t_up = sim_get_NNT_up_bound(outbuff);
: : :       prot_remove_from_list(p,&outmessagelist); prot_insert_in_buffer(packet,OB[outbuf]);
: : :      }
: :       goto S2_4; /* optional to process all local events until LVTW */
: : :    } /* while ( ! stop ); */
: : :  }
```

5 Conclusions

We have presented the strategy that we have followed for the simultaneous implementation of three different distributed discrete event simulation protocols in order to preserve performance comparability. For the widely demanded "fair" performance comparison of DDES protocols we have demonstrated at a sufficient level of detail how to exploit common characteristics of the different strategies. From our formulation of the logical process behavior one may realize that the efficient implementation of the optimistic and the conservative protocols require (after a proper restructuring of the code) a subset of the functionalities needed for the efficient implementation of the probabilistic protocol that we proposed. Exploiting these characteristics we were able to specify the three protocols at once using the optional compilation facility provided by the C language preprocessor right from the beginning. The identification of *shared* data structures and pieces of code was essential for establishing an evaluation that separates performance causes which are protocol related from those ones that are common to the protocols being compared.

Besides this, the approach is also beneficial from other points of view. First it yields a substantial amount of code reuse in the development of independent prototypes, i.e. once the development of the probabilistic protocol is complete, we obtain a (almost completely) debugged and optimized implementation of the other two protocols for free. Second, optimizations of the implementation can be handled more easily. If an optimization concerns all the three protocols, it is implemented only once and automatically imposed on every protocol. If an optimization is protocol specific, then its impact on the performance can be identified independently of the other protocol implementations. Finally the simultaneous optimization of all the three protocols is in favor of the comparison against the best sequential simulation implementation that one can use. The latter is a non trivial aspect of the implementation of such prototypes: it is not unusual to find distributed prototypes that are relatively better than other distributed prototypes, but that perform poorly compared to the best sequential implementations due to the substantially lower effort devoted to the fine tuning of the code efficiency rather than any specific reasons related to the distributed protocol in itself.

References

[1] K.M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, 5(11):440–452, September 1979.

[2] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(4):198–206, April 1981.

[3] G. Chiola, M. Ajmone Marsan, G. Balbo, and G. Conte. Generalized Stochastic Petri Nets: A Definition at the Net Level and its Implications. *IEEE Transactions on Software Engineering*, 19(2), February 1993.

[4] G. Chiola and A. Ferscha. Distributed Discrete Event Simulation of Timed Transition Petri Nets. *IEEE Parallel and Distributed Technology*, 1993.

[5] G. Chiola and A. Ferscha. Distributed Simulation of Timed Petri Nets: exploiting the net structure to obtain efficiency. In 14^{th} *International Conference on Application and Theory of Petri Nets*, Chicago, Illinois, June 1993. Springer Verlag.

[6] G. Chiola and A. Ferscha. Exploiting Petri Net Model Structure to Improve Distributed Simulation. In 26^{th} *Hawaii International Computer Science Symposium*, Honolulu, Hawaii, January 1993.

[7] G. Chiola and A. Ferscha. Self-Adaptive Logical Processes: the Probabilistic Distributed Simulation Protocol. Technical report, Austrian Center for Parallel Computation, Technical University of Vienna, 1994. submitted for publication.

[8] R.C. De Vries. Reducing Null Messages in Misra's Distributed Discrete Event Simulation. *IEEE Transactions on Software Engineering*, 16(1):82–91, January 1990.

[9] R. M. Fujimoto. Performance of Time Warp under Sythetic Workloads. In D. Nicol, editor, *Proc. of the SCS Multiconf. on Distributed Simulation*, pages 23 – 28, 1990.

[10] R.M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[11] A. Gafni. Rollback Mechanisms for Optimistic Distributed Simulation Systems. In *Proc. Conference on Distributed Simulation 1988*, pages 61–67, California, 1988. Society for Computer Simulation.

[12] A. Gupta, I. Akyildiz, and R. Fujimoto. Performance Analysis of Time Warp With Multiple Homogeneous Processors. *IEEE Transactions on Software Engineering*, 17(10):1013 – 1027, 1991.

[13] D. Jefferson and P. Reiher. Supercritical Speedup. In A. H. Rutan, editor, *Proceedings of the* 24^{th} *Annual Simulation Symposium, New Orleans, Louisiana, USA, April 1-5, 1991.*, pages 159–168. IEEE Computer Society Press, 1991.

[14] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. In P. Reynolds, editor, *Proc. Conference on Distributed Simulation 1985*, pages 63–69, La Jolla, California, 1985. Society for Computer Simulation.

[15] D.A. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[16] F.J. Kaudel. A Literature Survey on Distributed Discrete Event Simulation. *Simuletter*, 18(2):11–21, June 1987.

[17] P. Konas and P.C. Yew. Parallel Discrete Event Simulation on Shared Memory Multiprocessors. In *Proc. 24^{th} SCS Annual Simulation Symposium*, pages 134–148, New Orleans, Louisiana, April 1991. IEEE-CS Press.

[18] Y.B. Lin and E.D. Lazowska. Exploiting Lookahead in Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):457–469, October 1990.

[19] Y.B. Lin and E.D. Lazowska. A Study of the Time Warp Rollback Mechanism. *ACM Transactions on Modeling and Computer Simulation*, 1(1):51–72, January 1991.

[20] B. Lubachevsky, A. Weiss, and A. Shwartz. An Analysis of Rollback Based Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(2):154–193, April 1991.

[21] J. Misra. Distributed Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[22] D. M. Nicol. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *JACM*, 40(2):304 – 333, April 1993.

[23] D.M. Nicol and S. Roy. Parallel Simulation of Timed Petri Nets. In B. Nelson, D. Kelton, and G. Clark, editors, *Proc. 1991 Winter Simulation Conference*, pages 574–583, 1991.

[24] A. Prakash and R. Subramanian. Filter: an Algorithm for Reducing Cascaded Rollbacks in Optimistic Distributed Simulation. In *Proc. 24^{th} SCS Annual Simulation Symposium*, pages 123–132, New Orleans, Louisiana, April 1991. IEEE-CS Press.

[25] M.L. Yu, S. Ghosh, and E. DeBenedicts. A Provably Correct, Non Deadlocking Parallel Event Simulation Algorithm. In *Proc. 24^{th} SCS Annual Simulation Symposium*, pages 100–111, New Orleans, Louisiana, April 1991. IEEE-CS Press.