A Survey on Parallel Logic Simulation*

Gerd Meister

Department of Computer Science, University of Saarland
P.O. Box 1150, 66041 Saarbrücken, Fed. Rep. Germany

Email: meister@cs.uni-sb.de

September 21, 1993

Abstract

Simulation of logic designs is a very important part of the VLSI-design process. The increasing size of the designs requires more efficient simulation strategies to accelerate the simulation process. Parallel logic simulation seems to be a promising approach in this direction. This paper describes the basic principles of parallel logic simulation, discusses different approaches, and surveys the research done in this field so far.

1 Introduction

In recent years the increasing demand for fast development of integrated circuits has caused many studies in the field of design of microelectronics and hardware units. A particular problem arises from the exploding number of transistors that can be placed on single chips. Design methodologies such as placement, floor planning, channel routing [SES85a] are one major topic in research. Another and even more important theme is verification of such large designs. "Is the final product really going to show the expected behavior?" is one of the most important questions in this context. For this reason the simulation of the behavior of the planned chip is essential in the design process to avoid the fabrication of faulty chips.

In the past years, several rather different approaches have been made to this topic ranging from dedicated hardware which was especially built to support the needs of logic simulation (such as hardware support for event queue management) to various classes of software simulators for logic simulation.

In the next section of this paper the basic properties of logic simulation will be described. In Section 3 we will consider some specific characteristics of logic simulation

*This work has been supported by the German Science Foundation (Deutsche Forschungsgemeinschaft), project D1b, SFB 124

that are important when one considers parallel simulation of digital circuits. Section 4 describes different approaches to realize parallel simulators. After this we will point out which work is currently done in this field (Section 5). Section 6 is dedicated to closely related topics, in particular the partitioning of circuits, load balancing in parallel applications, and the dependencies between computation and communication overhead.

A partial overview on parallel logic simulation is also given by Briner and Soulé in [BRI90a] and [SOU92a], respectively. However, the authors focus on their own work and do not treat this theme in a general way. A more comprehensive overview (in german) is given in a recent report by Rössig [ROE93a], where the author surveys parallel logic simulators and compares the underlying hardware, the strategies, and the partitioning aspects.

2 Principles of Logic Simulation

Simulation of digital logic circuits is mainly used to avoid serious design errors that may render the costly development process of a hardware unit worthless. This is even more important as the development may take several months or even years. Another important task of simulation is to verify whether a design fulfills its specifications.

The simulation of a planned design can discover many of the problems that may arise. Usually, the chip design goes through several iterations. Starting with the initial design, the prototype has to be checked using design rule checkers for electrical behavior and simulation for functional tests. If errors are detected, they have to be corrected and another iteration is performed until the specified design is realized. Chip design can be supported by hardware description languages which describe the modeled unit in a programming language-like fashion. Examples are VHDL (Very High Speed IC Hardware Description Language) [VHD87a] and Mimola [ZIM80a, JOM89a].

The specification in terms of such a language can also be used as input to the simulation tools. Hardware description languages provide a complete specification and documentation frame for the development of circuits.

There might be different ways to describe the chip. VHDL, for example, provides three views on a model:

- a behavioral description,
- the structural interconnections of the basic components, and
- the logical functions implemented by the unit under design.

Additionally, one may combine several views arbitrarily within one model. Hardware description languages provide an abstract view of the chip and help to obtain more structured and comprehensible designs.

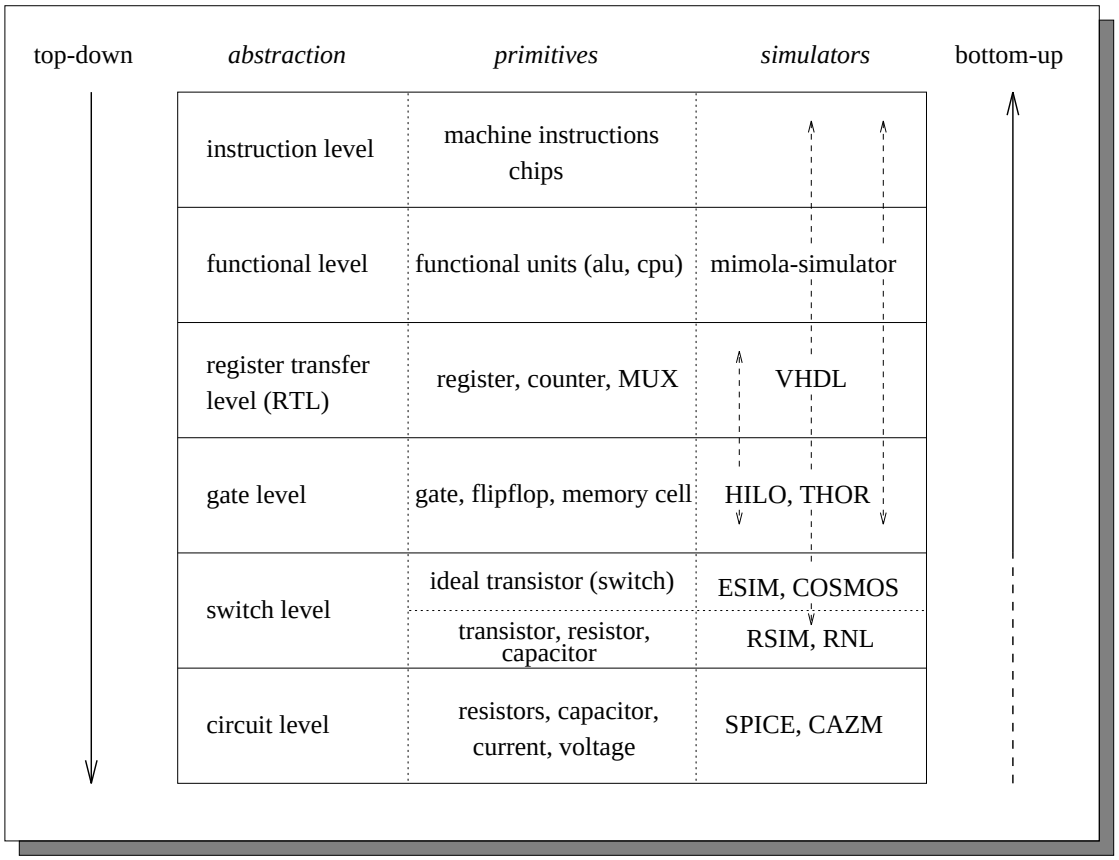

Figure 1: Design Hierarchy and Abstraction Levels for Logic Design

2.1 Design Abstraction Levels

Usually, the design of a chip will not be done totally from the scratch. Instead, the trend goes towards the reuse of building blocks such as elements of cell libraries for logical gates or register transfer primitives. Such components are obtained from earlier designs or from specific libraries to support a faster design process. The elements of such libraries can then be composed to form new circuits.

Even then, however, it is often necessary to simulate the new design at a higher level to verify the interaction of the preconstructed units. For this reason, it is obvious that simulations are performed at very different layers. One may start at the top level description of a whole system, and refine the model at the register transfer level (RTL). A further level of detail consists in the simulation of gates, switches, or transistors. This process of decreasing abstraction is known as top-down design. The opposite approach is the bottom-up design where libraries are used to compose complex objects from already existing simpler devices (see Fig. 1).

In the bottom-up approach, the most accurate simulation is done by an analog simulator. The next step towards abstraction is the consideration of digital circuits that can be modeled at the switch or transistor level. The description of the timing

behavior of an analog circuit through differential equations is replaced by dynamically calculated switch times from a capacitance/resistor model. The next higher layer views transistors only as switches that react on their inputs with a given fixed delay. Here, only the delays are of importance while exact switching times due to the actual number of driven fanout elements are neglected.

On top of the switch layer we can find the gate level grouping together several transistors to logic gates, memory cells, and flip-flops. These elements may be combined to RTL primitives (MUX's, counters, or registers) in a next step. Another layer consists of functional descriptions of complete units such as ALUs or CPUs composed from RTL building blocks and simpler elements. The top of the hierarchy is built by instructions that realize complex operations of a system such as a load or store instruction for a CPU. This level is mainly used for the simulation of complex logic circuits. However, each level of abstraction has to be paid with a loss of information on the exact behavior of the simulated circuit. For this reason, almost every design process has to be verified at one of the lower levels to ensure the desired behavior of the final product.

Another often used principle is to combine the elements of several levels within one single model and to perform a mixed-level simulation. This is mainly applied to elements of the switch level, the gate level, and the register transfer level. However, all elements of a mixed-level simulation show digital characteristics. If analog simulation is also incorporated into a multi-level simulation process, we speak of multi-mode simulation where analog and digital behavior of the simulated elements is mixed.

2.2 Timing Models

In logic simulation several possibilities to model the temporal behavior of circuits are used:

- *Continuous time* is mainly applied for analog simulation of the lowest level of the circuit. The currents and voltages are expressed through differential equations in dependency of the time.

- *Unit delay* assumes that every change of a signal's value needs exactly one time unit to become available. This means that a change at an input of an element is reflected in the next step on the output of that element.

- *Fixed delay* assigns individual delays to each element which are kept constant throughout the whole simulation. By this, different falling and raising times can be modeled and the circuit can be simulated more accurately.

- *Variable delay* provides the most flexible way to simulate elements. It is used in particular at the lower levels and allows different switching times due to changing fanouts or capacitances that depend on the actual state of the simulated system.

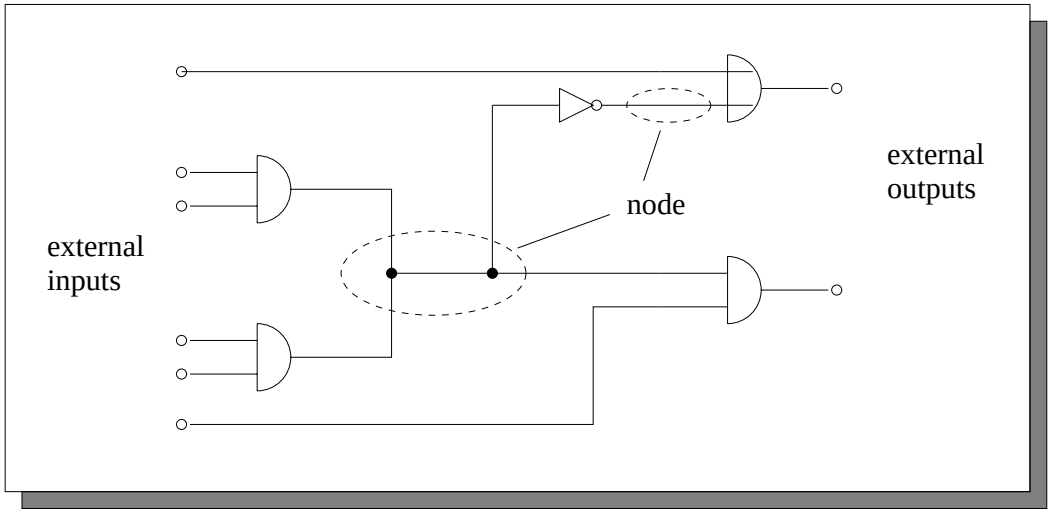

Figure 2: Simple Schematic of a Model at the Gate Level

2.3 Environment and Structural Model for Logic Simulation

The behavior of a design is simulated by feeding the description of the circuit obtained during the development into a simulator. To actually run the simulation, one also has to supply the simulator with input data (*stimuli*). The stimuli might be randomly generated or extracted from the specification. The results on the outputs of the simulated circuit are evaluated to see whether the design shows the specified and expected behavior in functionality and speed.

The basic actions that are performed during logic simulation are identical for all abstraction levels and discrete timing models. For this reason, we refer to any element of a circuit that is contained within a model as a *unit*. In general, each unit is provided with at least one input and at least one output and has an internal state that is time-dependent. The inputs and outputs of the basic units are interconnected through wires over which signal values are propagated. A signal value changes at discrete points in time. The wires are realized within a simulator as *nodes* that store the current signal value of the wire and possibly future signal changes. A wire's signal may be influenced by multiple units that are connected to it. For this reason, the node has to separate the values supplied on the outputs of the driving units from the inputs of the driven units as shown in Fig. 2. This is necessary because

- several units driving a node with their outputs may supply different values at the same time and
- the values of the inputs and outputs must be maintained until all units are evaluated at the current time. A new value provided on an output for a future time must not be delivered to any input before its associated units have been evaluated for the current timestep.

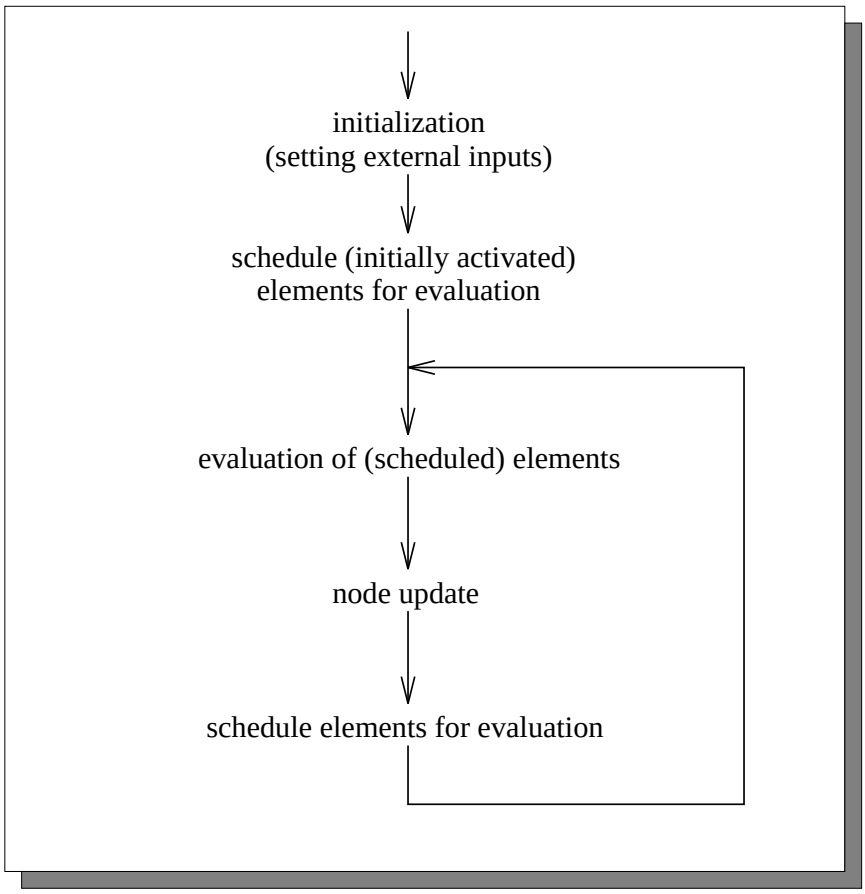

Figure 3: The Basic Simulation Procedure

Thus, to separate input and output behavior, nodes must be inserted. They may also provide so-called resolution functions. If there are multiple driving units that may influence the signal values simultaneously, a resolution function determines the values that will be propagated to the inputs of units that are driven by this signal.

For simplicity of the model, the external inputs of the units (e.g., the pins of a chip) are considered as nodes which are driving the connected units by supplying them with the current values of the external links.

2.4 The Basic Simulation Scheme

The simulation of a circuit is performed as a cyclic operation preceded by an initialization phase before entering the loop (Fig. 3). In the initialization phase, the a priori known stimuli are provided to the inputs of the circuit. After this, all units that may be affected by the initial input settings are scheduled for evaluation of their internal states. Then the loop is entered.

Each iteration of the loop is composed of three steps. In the first step, the new states of the scheduled units are calculated. Changes of states possibly result in

changes of the outputs of the units which are driving inputs of other units via the intervening nodes. If this is the case, the new values of the nodes are determined in a second step according to the resolution function when all values of their inputs are available. In the third step, the simulator schedules units for reevaluation. In particular, units which are driven by nodes that changed their values in the previous step have to be evaluated.

The loop will only be left when no more changes take place. This happens when the model becomes quiescent or if a predefined termination condition is reached (e.g., when the simulation time exceeds a limiting value).

The data structures for the realization of such a simulator generally consist of a list of units with their inputs, outputs, and current states, and of a list of nodes including the driving and driven units. Often, an event queue that is used to schedule units whose inputs changed is also part of the simulator. During each cycle of the loop described above, the appropriate operations on the units, the nodes, and the event queue are performed in order to update the data structures according to the simulation model.

2.5 Sequential Simulation Strategies

The main criterion to distinguish between different simulation types is the way how simulation time behaves and is advanced. According to this, we can partition the simulation methods in two classes:

- **Time-Driven Simulation** can be viewed as slow-motion or quick-motion of the simulated system. Based on a given discretization of time, all timesteps are simulated. Time-driven simulation is also known as *compiled mode simulation* because the strategy after which units are evaluated for each step is determined once at compile time of the simulator and cannot be adapted during simulation according to dynamic requirements.

  This method is the most simple way to do a simulation. The simulation time usually advances in equidistant steps. For each step all units are evaluated. The new state values are stored in a second set of state variables that will take the role of the old state variables in the next cycle. The third step of the simulation loop of the general scheme degenerates to the scheduling of all units. However, this scheme may be optimized by evaluating only those units for which the input signals changed in the last step. This optimization is also called synchronous event-driven scheme.

  The size of the simulation timestep depends on the desired accuracy of simulation. For fixed and variable delay it has to be chosen in a way that the execution order in the real system is reflected by simulation time, i.e., two activities with different execution times in the circuit are modeled in two different simulation timesteps. At the end of each cycle, the simulation time is increased by the step size.

- **Event-Driven Simulation**: The main disadvantage of time-driven simulation is that in the case of many idle time intervals (with no changes or activities at all) much execution time is wasted in evaluating units that did not change, and in checking whether something has changed or not. This drawback is avoided by the event-driven method which models the changes of signals as events with associated discrete points in simulation time, so-called timestamps. Timestamps indicate when the events are to be executed.

  To maintain the temporal order of events in the event-driven simulation, the events are always simulated in increasing timestamp order. Idle times are skipped which typically results in faster running simulations. The properties of event-driven simulation will be detailed in Section 4.

3 Parallel Logic Simulation

3.1 Aspects of Parallelizing Logic Simulators

As the size of the designs to be simulated rapidly grows, the need for fast simulation tools cannot be satisfied any more by sequential simulators on general purpose machines. The gap between the speed required by the user and the speed delivered by available simulators grows. There are two solutions to this problem:

- The first approach consists in building special simulation hardware such as the Yorktown Simulation Engine YSE [PFI82a] or the MuSiC computer [HAF85a]. These machines consist of several special purpose processors which support fast synchronous time-driven simulation or dataflow-driven simulation. The problem of such specialized processors is that they are very expensive and that they often only support a restricted class of models. The YSE, for example, can only deal with gates with four inputs. Hence, the design process is severely restricted by this approach. Mixed-level simulation seems to be impossible with such restrictions.

- The second approach consists of executing the logic simulation software in parallel on a general purpose multiprocessor and thus exploiting the parallelism of the models to be simulated.

The second strategy will be detailed for digital discrete event simulations in the following sections.

3.2 Metrics for Parallelism

Parallelization of logic simulation tries to exploit the computational power of multiple processors in a multi-computer environment by concurrently executing the model under simulation. To achieve this, several processors are performing the work in parallel

which a single processor would have to perform in a sequential order. The underlying idea is that there should be enough activities to simulate in parallel at every instance in time during the simulation, so that the unavoidable overhead involved with parallel simulation is paid off. One important problem is how to define and to measure possible parallelism.

The general potential for parallelization in time-driven simulation has been examined in [WFC86a] and [BAI92a]. There it is described that models derived from realistic designs show only a relative activity of 1.2 %. This means that by averaging over all nonidle times, less than 2 % of all units in the model are evaluated. If the calculations of the relative activity also include idle timesteps, the activity values fall below 1 %. Additionally, the relative activity does not scale linear with the circuit size as one might expect. Instead, with increasing design sizes, the potential for parallelism from units that can be evaluated simultaneously grows slower than the dimension of the circuits.

This leads to the question how an appropriate measure for concurrency and potential parallelism has to be defined. In the two papers cited above, the average percentage of active units over nonidle timesteps is used.

Another measure has been proposed by Briner [BRI90a]. This measure was derived from an asynchronous event-driven approach for parallel simulation. For this idea, the model is distributed onto several sequential simulators which simulate only their assigned portion. Each simulator is free to proceed at its own speed, and therefore different processors may possess differing simulation times.

Briner suggests to take into account all units that may be evaluated at a given instant in time even if the corresponding events have differing evaluation times. The basic idea is that concurrency is determined by the number of events that are simultaneously executable based on a real time approach instead of simulation time as in the previously described scheme. This method, however, strongly depends on the progress in simulation time of the single simulators. For this reason, the measurements of different simulation runs may vary which makes it difficult to draw the right conclusions concerning the circuit behavior.

3.3 Distribution of Data

Todays parallel computing environments are mostly either SIMD or MIMD architectures [HWB84a]. A SIMD architecture, where the same operation is executed on different data (e.g., a vector) at a time, does usually not provide the flexibility needed to simulate logic circuits in an appropriate way. In logic simulation the units often show quite different behavior so that SIMD machines are not well suited for this task. In the following, we will therefore assume the availability of MIMD architectures as a basic hardware for the simulators.

There exist two large classes for these architectures: shared memory and distributed memory machines. Shared memory systems may hide the differences between the access methods to local and remote data, whereas the processors in a distributed

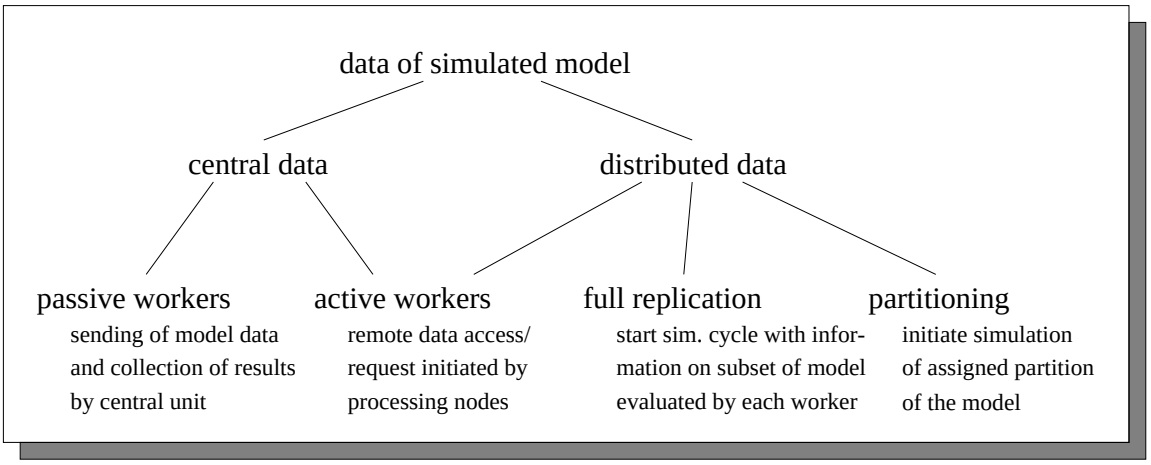

Figure 4: Data Access and Distribution Possibilities

memory system can only access their local memories, and data between processes must be exchanged via messages.

Access to data of the simulated model can be provided in different ways (Fig. 4):

1. The whole information is stored on a central server and portions of the data are assigned dynamically to worker processes for execution of the appropriate operations. After processing, the results are retransmitted to the central server. This scheme can be modified to a demand-driven method where idle workers may request supplementary work from the central server.

2. The data is replicated on each processor and a centralized or decentralized algorithm decides which portion is actually simulated by each worker.

3. The data is partitioned into subsets which are distributed to dedicated worker processors. The subsets comprise disjoint parts of the whole model and form submodels. Each worker operates only on the assigned subset, i.e., it performs only the evaluations for the units within this subset.

Applying these methods on either shared or distributed memory systems shows various strengths and weaknesses of the approaches.

The first scheme will perform quite well in a shared memory environment where all processors can access all available data. The load (i.e., the work that is performed by the worker processors) is implicitly balanced. Each worker will work on the problem as long as there is something to be done. In a system with distributed memory, load balancing is a non-trivial problem.

The second strategy involves additional overhead to keep the replications in a consistent state but, compared to the first scheme, it avoids the sending of huge amounts of data when a distributed memory system is used. The distribution of the units' data through messages might become a problem especially for mixed-level

simulation where units may have large states. For shared memory there might arise another problem if shared data structures, such as an event queue that is accessed by all worker processors to get work, are referenced too frequently. Again, as in the first scheme, load balance can easily be established.

The third strategy is often applied in distributed memory environments and is used to keep communication costs low. It may, however, also be used for simulation of large models which do not fit into the memory of a single processor. If this scheme is realized in a shared memory system, one may enhance the method combining the possibility of being able to access remote data for load balancing in an easy manner with the advantage of keeping as much computation as possible locally on the processors. This is also a main problem in partitioning and load balancing which will be discussed in Section 6.

Another criterion beside the memory paradigm is the question whether simulation is based on the time-driven or on the event-driven method. For event-driven simulation, the third strategy reflects the basic behavior of this kind of simulation quite well because the simulators on the processors communicate by sending messages. Additionally, the simulation times that represent the state of simulation can differ among the simulators running on the processors. The two other approaches can easily be used with time-driven simulation with a central or decentralized control that gives a starting shot for each parallel evaluation cycle and synchronizes at the end of the cycles. The management of units with differing states of progress in simulation time might involve too much overhead in these cases.

4 Methods of Parallelizing Logic Simulation

In the following sections we will point out which approaches for parallelization of logic simulation are possible. The parallelization techniques can be divided into two classes: *synchronous* and *asynchronous* schemes.

4.1 Synchronous Parallel Simulation Schemes

For the synchronous schemes, all parallel instances operate closely coupled at the same simulation time. These methods use a central clock as a time base for all processors. An example for this strategy is the time-driven algorithm described in Section 2. The evaluation of units is performed in parallel by distributing the set of scheduled units onto several processors. After completion of the evaluation step, the states of the units and the signals are updated and the set of events for the next cycle is determined. The units for which new events have been created are scheduled for reevaluation. This approach is depicted schematically in Fig. 5.

One recognizes that the central time causes the operations to be synchronized after each step. Therefore, the speed-up that can be achieved depends on the evaluation speed of the single units. The slowest unit determines the speed of the simulation

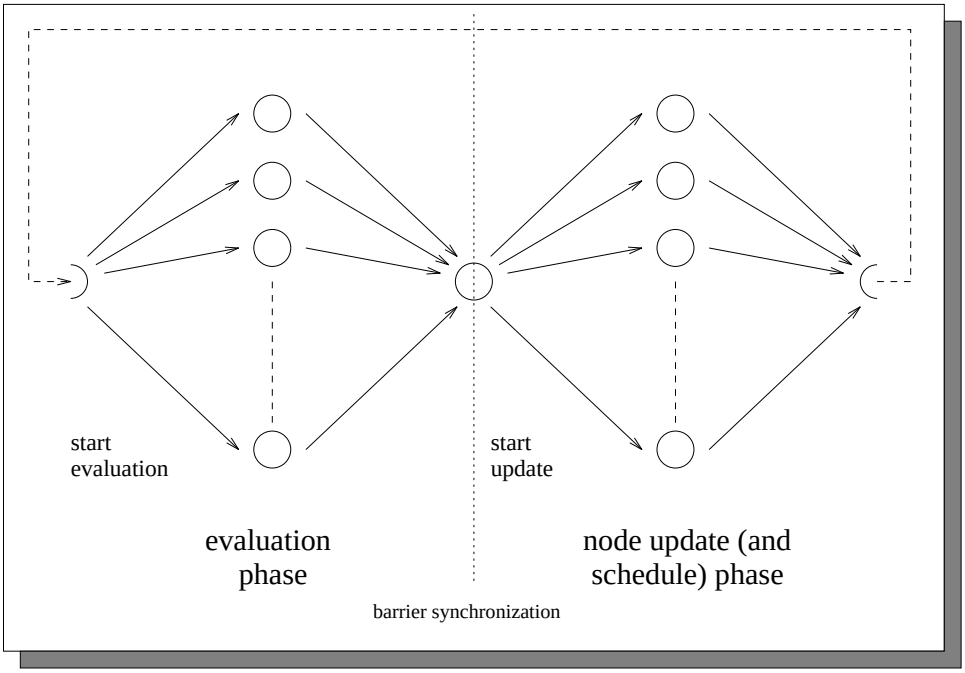

Figure 5: Synchronous Lock Step Parallelism for Logic Simulation

time advancement with respect to real time. This might be a problem especially in mixed-level simulation where a model may contain simple gates, as well as a whole ALU or CPU as units.

4.2 Asynchronous Parallel Simulation Schemes

A quite different starting point is the introduction of asynchrony in parallel simulation. The basic mechanism is discrete event-driven simulation. For this approach, the model is partitioned into disjoint submodels. Each processor is charged with the simulation of one or more submodels. The simulation of each submodel is performed by a so-called *logical process* (LP for short) which behaves like a sequential simulator and may interact with other LPs if local changes influence units beyond its own submodel.

Each LP consists of a local state comprising the data concerning the local partition of the model. All actions that are performed in the submodel are realized as events which have an execution timestamp and may cause the local state to change. Additionally, an LP has a local clock which proceeds at its own speed and which depends only on the local events that must be executed in increasing timestamp order (Fig. 6). There exists no notion of a global clock or global state. The local clocks of different LPs usually differ and the LPs run in asynchrony.

To maintain the timestamp order of the executed events, all events are kept in an event queue in increasing timestamp order. The LP always picks the event with the smallest timestamp from the event queue and executes it. The processing of an event

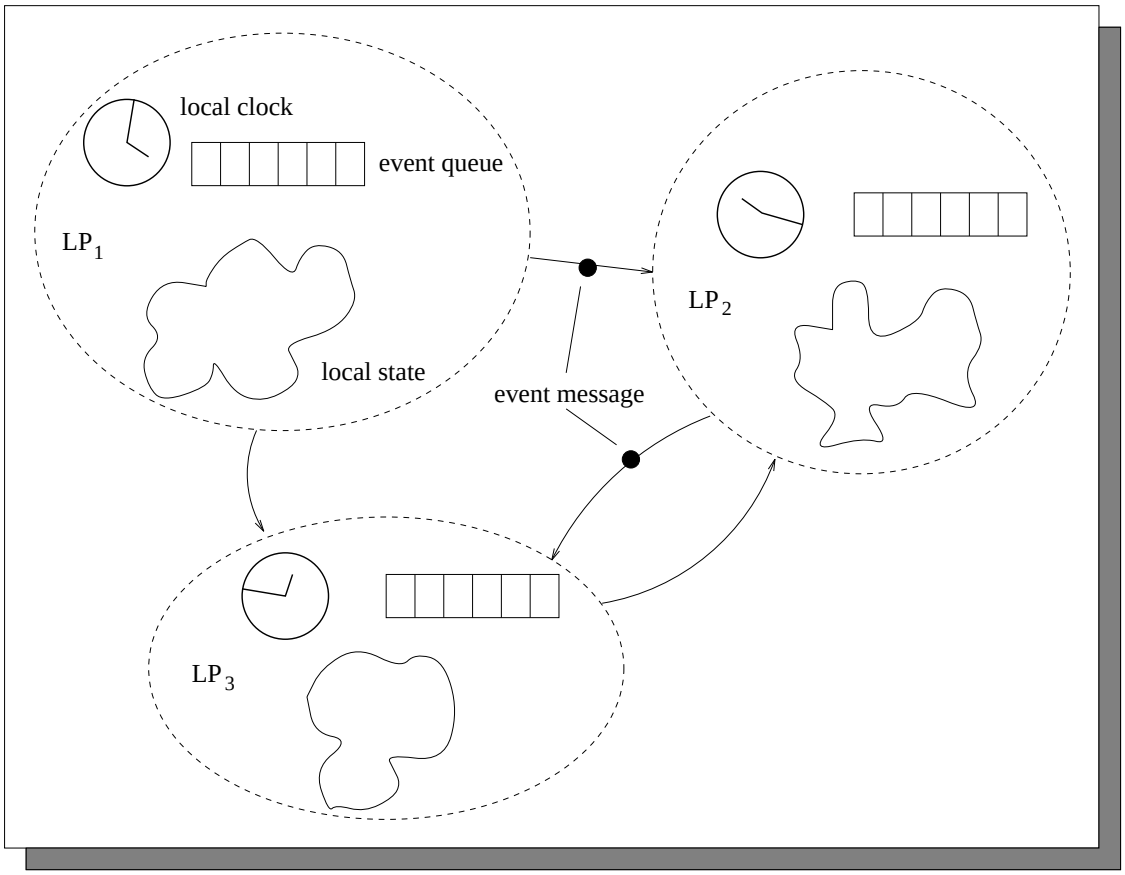

Figure 6: Structure of a Distributed Simulator

advances the local clock to the execution timestamp of the event and may create several new events which are to be executed in the future. If the a new event affects a unit of the local partition, it is simply enqueued in the local event queue.

Events that were generated for non-local units, so-called *remote events*, are sent via an *event message* to the LP that is responsible for the simulation of this unit. On the reception of an event message, an LP creates a local event that reflects the effects of the remote event on the destination partition, and inserts the new event into the local event queue. To maintain the correct timestamp order of the local and remote events, each event message contains a timestamp that indicates the local time of the sender at which the message was created and a timestamp that tells the receiver when the event has to be executed. This whole procedure is usually called remote event scheduling.

In the case of parallel logic simulation, the events reflect changes of signals that drive units or may be driven by units. As signals are generally realized as nodes, signal changes that affect other partitions must also be propagated through the nodes. This implies that the nodes themselves may be distributed if their inputs and outputs connect different LPs (Fig. 7). Thus, the distributed nodes are creating event

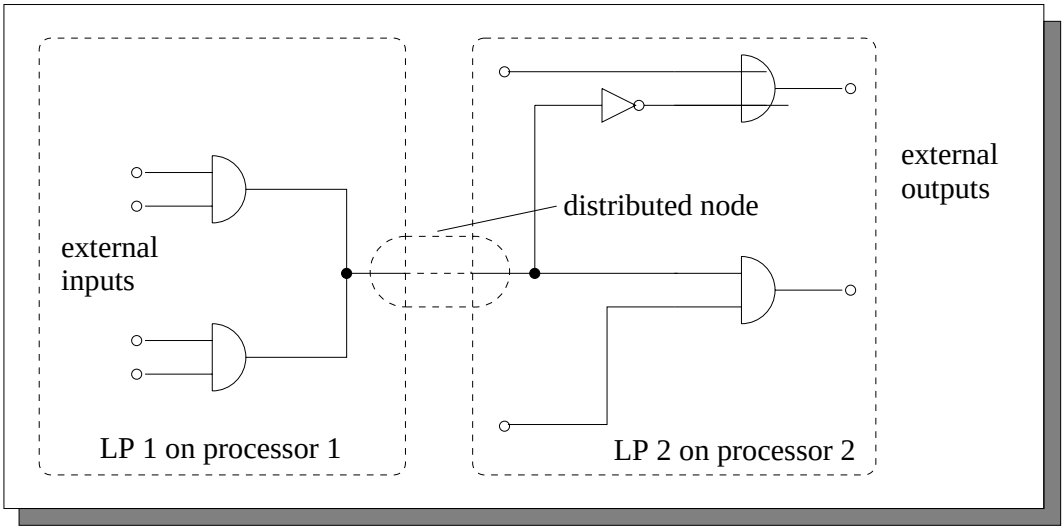

Figure 7: Schematic of a Distributed Model at the Gate Level

messages to propagate the signal changes to their remote portions.

We generally assume that a node and all units that feed it reside on the same LP. If nodes and feeding units were placed on different LPs, additional work would have to be done to maintain the states of the nodes through event messages. Hence, the nodes were no longer auxiliary constructs that are used to separate evaluation cycles but they would have to be handled as a special type of units for signal propagation instead.

The way how distributed nodes and event messages are realized depends on the programming paradigm (i.e., shared memory or message passing).

4.2.1 Synchronization of Logical Processes

The absence of a central simulation clock gives raise to the problem of synchronizing the LPs. In a sequential simulation all events are executed chronologically. Furthermore, unless there is explicit nondeterminism, two runs of a sequential simulator will always produce identical results for the same model. This is a behavior which a user of a parallel simulator expects too.

In a sequential simulator no event with execution time (timestamp) t_1 can be executed before an event with execution time t_2, if $t_2 < t_1$ holds. For sequential simulation, the order in which the events have to be scheduled according to their timestamps is always respected. This means that all events e_i that may influence an event e_j are executed before the execution of e_j. Because of this property, the order and dependency of actions in the real system, which is the base for the simulated model, is reflected. For events with identical timestamps a deterministic resolution scheme has to be used to guarantee deterministic reproducibility [MEH91c]. For simplicity we assume here that no two events with the same timestamp appear in the simulation.

In a parallel environment where each LP maintains only its own local simulation clock and its local state, global states are generally not available. For this reason, it is difficult for the LPs to predict whether another LP will schedule a remote event for them with an execution timestamp that is smaller than the smallest timestamp of all events in the local event queue. Of course, such an event must be executed before the locally known events to ensure that the timestamp order is respected. The decision whether to proceed or whether to wait for possible remote events has to be made only with the local state information available to the LPs.

Because of the lack of global knowledge, one has to take appropriate means to assure that all events are simulated in the "right" order and that the causality among the events is maintained. There are two main principles to achieve this which will be sketched next.

4.2.2 Conservative Approach

In conservative approaches, a local event is only processed if the LP can be sure that no late events (i.e., events that have a timestamp which is smaller than the LP's local clock) may arrive anymore. This may be achieved by associating logical channels between those LPs that may schedule remote events for each other. Each time an LP schedules a remote event for another LP, the channel's clock is set to the value of the originating LP's local clock. Because each LP only processes events in increasing timestamp order, and a remotely scheduled event always has a higher timestamp than its originating event, the values of the channel clocks cannot decrease[1]. Processing an event is always preceded by checking the clocks of the incident channels against the timestamp of the event. If the event possesses the smallest timestamp among all those values, it is safe because the LP knows that no late event messages can arrive and the event can be executed. In the other case, late events may arrive and the LP has to block until all channel clocks satisfy the required condition.

The blocking property of the conservative methods almost always causes deadlocks to appear during simulation of models that contain feed-back loops. To solve this problem, the LPs have to gather supplementary information to detect and resolve deadlocks, or they have to use methods which avoid deadlocks (e.g., so-called null-messages). The deadlock problem sometimes causes serious overhead for parallel distributed simulations and the general usefulness of this approach is still unclear. A general and more detailed description of the conservative methods may be found in [CHM81a, FUJ90a, MIS86a].

[1]For the case that the channels do not preserve delivery of event messages in FIFO-order, supplementary actions must be introduced to prevent decreasing channel clocks that would disturb the conservative methods. Appropriate means for dealing with non-FIFO channels are, for example, counting mechanisms which record the number of event messages that are still in transit.

4.2.3 Optimistic Methods

The second class of simulation algorithms is called optimistic. Each LP is free to execute in timestamp order all events which are currently in its event queue. Only when a late event, a so-called *straggler*, arrives, then the local clock that represents the virtual time of the LP and the state of the LP are reset to a state that guarantees that the straggler and the other events can be processed in the right order [JEF85a]. Resetting an LP is called *rollback* because the local states and clocks are rolled back to former states and clock-values. Event queues are also rolled back, and the affected events have to be executed again after the execution of the straggler.

For this reason, each LP periodically has to save its local state to be able to restore it if a rollback occurs. As possibly erroneous calculations were performed because of a straggler, all effects of those computations must also be reset. This cannot always be restricted to local recovery. Often it is necessary to cancel also the effects that were caused on other LPs.

The best known representative of the optimistic methods is *Time Warp* [JES85a]. Time Warp was designed for an environment where several LPs interact by sending event messages. During a rollback the effects on remote processes are revoked by sending *antimessages* for remote events that became invalid. The LPs keep the event messages in an input list until they are processed. If an antimessage finds its corresponding event message still enqueued in the input list, both messages are annihilated. If the event message has already been processed, a secondary rollback for the receiving LP has to be done. This might result in cascades of rollbacks. There exist several optimizations that avoid a too deep level of rollbacks which might render the whole method worthless [FUJ90a].

It should be noted that the time spent in speculatively processing events by an optimistic method would be "wasted" by the conservative strategies in blockings caused by waiting on new events or by deadlocks and their resolution.

4.2.4 Conservative or Optimistic?

Much work has been done to learn which approach is best suited for typical classes of simulation applications. Lin, Lazowska and Bailey defined an optimal conservative method which is based on simplified assumptions on the behavior of conservative protocols. This scheme was compared to Time Warp and several conservative methods. From their theoretical examinations, they established a hierarchy with Time Warp being superior to the optimal and the other conservative algorithms [LLB89a].

Reynolds lined out that the distinction between optimistic and conservative methods is not sufficient because there exist many mixtures in existing strategies that cannot be assigned to either of the two methods [REY88a]. Many possibilities for synchronizing distributed simulation have not yet been investigated. Therefore, he designed a testbed which provides support for an easy and fast implementation of different methods. This testbed allows also the realization of multiple applications. The experiences from this project show that it is not quite easy to evaluate the dif-

ferent approaches and that much more conceptual work has to be done to obtain a more clearly classification or ordering of synchronization schemes.

Schemes that dynamically adapt their behavior either to be more optimistic if the situation allows this, or to return to a more conservative method if too many rollbacks occur, have also been devised. The practical studies and experiences on parallel logic simulation are, however, dominated by the two basic approaches.

5 State of the Art

In this section we will give an overview on current and earlier work performed by research groups on parallel logic simulation. It should be noted that much conceptual and practical work on parallel and distributed simulation has also been done outside the specific application area of logic simulation. A survey on this work is out of the scope of this paper, although some of the general results are of course also of relevance to logic simulation. The interested reader is referred to [FUJ90a] where important contributions to the general theory of parallel and distributed simulation are surveyed and discussed, and further pointers to the literature are given.

5.1 Theoretical Studies and Conceptual Work

5.1.1 Wong et al.

In [WFC86a] the general possibilities to parallelize logic simulation are examined. According to the authors, the logic simulation process can be divided into several subtasks: event queue manipulation, functional evaluation of units, netlist operations (e.g., propagation of changes of signal values), data output, startup operations, and other overhead that does not fit into one of these classes.

They instrumented a sequential gate and switch level logic simulator, *lsim*, to gather statistics on the relative frequency of the subtasks mentioned above in discrete event simulation. The experiments of the authors showed that all of the above listed subtasks use approximately the same amount of time in the execution of a simulator and that consequently all of them have to be sped up to achieve reasonable acceleration of simulation runs by parallelization. This implies that there is no general bottleneck that might have negative impact on acceleration efforts and dooms to failure the approaches right from the beginning.

They also measured the total number of events associated with each of the circuit components, the size of the event queue, and the mean distance between the execution times of two succeeding events in the event queue. These experiments showed that in (synchronous) lock step simulation with unit or fixed-delay models there were on the average 86 % of idle timesteps with no events to simulate. The average number of simultaneous events at each nonidle timestep was in the range of 0.08 % to 3.1 % relative to the total number of gates in the model.

The authors tested five circuits from real designs with sizes between 650 and 7,950 transistors which were stimulated by randomly generated input. Additionally they checked the number of simultaneously enqueued events. In 90 % of all cases a maximum queue length less than 200 was observed.

5.1.2 Bailey and Snyder

In [BAS88a] the problems of the generation of stimuli for logic simulation and also the influence of circuit size on available parallelism are investigated. Bailey and Snyder followed Wong's approach in [WFC86a] for the measurements with six chips (approx. 200 – 27,000 transistors).

First, they examined the influence of the chosen time base on parallelism. Three different timing schemes were used.

For the first timing model, the size of the timesteps was enlarged by grouping together events from different time-slices into one interval without respecting causal relationships between them. This resulted in a linear increase of parallelism at the cost of lost accuracy and correctness of simulation.

The unit-delay model used in the second class of experiments respects causality and order of sequential execution and represents an upper bound for the third approach in which the queueing delay of the events is changed, i.e., the time which is needed to process a single event is raised artificially to reduce the influence of the communication overhead on the whole simulation time. The latter method also respects causality by simulating dependent events in different timesteps.

Bailey and Snyder found that increasing the timestep size usually causes a higher degree of parallelism according to the chosen metric (event queue length) because either more events are enqueued or the events are enqueued over a longer period of time. For their measures they selected the unit-delay model with a step size of 0.1 ns with which very accurate results can be obtained. The simulation results were compared to the universally accepted analog circuit simulator, *SPICE* [NAG75a], showing only very small deviations. From this fact the authors conclude that their measurements are general enough to be valid for a large class of logic simulators.

While recording the average activity rate within the tested circuits, (i.e., the number of units that are evaluated compared to the total number of units) they found that approximately 5 events are scheduled at a given simulation time resulting in an average activity rate between 0.04 % and 2.9 % scaled to the circuit size. It was also observed that the examined circuits have a quite sequential behavior with only one executable event for approximately 45 % of the simulation timesteps.

The circuits were first initialized by random stimuli to avoid faulty measurements due to start up phenomena. Afterwards, a second set of random input was supplied to perform the measurements for parallelism. For increasing circuit size, it was found that there is no obvious linear relationship between parallelism and size of the chips. The increase in observed parallelism stays sublinear.

The question whether Wong's measurements described in the previous section

are representative for larger circuits is also the theme of Bailey's more recent work reported in [BAI92a]. She simulated nine circuits with a unit-delay and a variable-delay simulator (*SwitchSim* [FRA85a], based on a hardware simulator, and *RNL* [TER83a]) at the switch level. The simulators were instrumented to calculate an event metric (the number of simulated events at an instant in simulation time) and a queue metric defined to be the number of scheduled events in the event queue. The circuit sizes range from 200 to 61,000 transistors and the chips were taken from real design processes.

Special emphasis is put on the scalability of concurrency within circuit families. These experiments have been performed for four of the nine circuits by using differently dimensioned instances of the same circuit types. Bailey found no clear tendencies whether the activity rate of a circuit scales well with the circuit size. It strongly depends on the type of the circuit as well as on the used timing model; hence further investigations are required and predictions from existing results are hard to make.

In particular the question whether there is a linear relationship between potential activity available for parallelization and circuit size is still not resolved because there are many factors that influence distributed simulation so that a clear decision on this topic is not yet possible.

5.2 Empirical Studies and Prototype Implementations

The previous approaches [WFC86a, BAS88a, BAI92a] were performed on modified and instrumented sequential simulators trying to predict the behavior of a parallel application by using supplementary calculations to obtain an estimation on the available parallelism. In this section the work of different research groups on actual parallel simulators will be summarized.

5.2.1 Soulé, Blank, and Gupta

Soulé and Blank examined the dependency between simulation duration and level of abstraction for given circuits [SOB87a]. Another point in this paper is the investigation of similar statistics as presented in [WFC86a] for average event queue lengths and average number of executable events per time. They categorized and analyzed four views of abstraction for logic simulation:

- instruction level,
- functional level,
- register transfer level, and
- gate level.

Their investigations were performed for three chip designs. Among them was a quite large multicomputer design consisting of approximately 150,000 gates. It was observed that for each level of detail the total execution time grew by a factor 10.

By examining the execution time needed by the different subtasks of the simulator (model evaluation, event processing, netlist updates) they found that for more detailed specification of the circuit, the netlist update phase needs more and more computation time. This is due to the fact that the number of signals in the circuit increases while the evaluation of the fine grained units can be performed in less time. For the activity rates and the queue lengths similar behavior as found in [WFC86a] was recorded. The activity rate is less than 1 % and there are relatively short event queues which contained in general less than 500 entries.

In [SOB88a], Soulé, and Blank compared an optimized parallel synchronous event-driven algorithm and a parallel compiled-mode algorithm to a parallel asynchronous event-driven simulator with a conservative strategy described by Chandy, Misra, and Bryant in [CHM81a, BRY77a] (CMB) which is essentially a deadlock detection and recovery strategy.

The experiments were performed on an Encore Multimax machine with shared memory. The circuits under test were several quite simple designs in the range of some thousand gates. Their results show that for all algorithms speed-up can be achieved although the speed-up value strongly depends on the type of circuit. A major distinction has to be made between circuits with feed-back loops and circuits without any feed-backs. Another factor that influences speed-up is the grain size of the simulated units. For a higher level of abstraction (e.g, RTL) the asynchronous version performs quite well. If the circuit contains feed-backs, the conservative asynchronous algorithm mainly has to detect and to resolve deadlocks.

The problems with asynchronous event-driven approaches are further investigated in [SOG89a] by Soulé and Gupta. The main topic of this paper is the problem how the asynchronous method is influenced by deadlocks that might appear in circuits with feed-back loops and cycles. Additionally, the authors examined some other features that decrease the expected acceleration of parallel simulation. In measurements with four realistic circuits they discovered that deadlocks are a main drawback for this type of simulation and that much higher speed-up values would be attainable if more efficient methods were available for deadlock handling. To reach this aim, deadlocks were analyzed and categorized into four classes according to their origins. The four classes contain deadlocks caused by:

- clocked registers that wait for the next clock tick while all other units have completed execution,
- paths with different delays (e.g., one input has to pass an inverter before it is input to another gate),
- order of node update, where a unit may be activated or not depending on the order in which the units are evaluated by the simulator, and

- unevaluated paths (e.g., there are signals that do not change for long times but which are supplied to units as inputs). Hence, no channel times are advanced for these signals and the simulation blocks.

For each class, solutions to prevent some of those deadlocks are proposed. Finally, Soulé and Gupta show that the classification covers almost all of the appearing deadlocks. For this reason, it seems plausible that higher speed-ups may be obtained if the proposed solutions are applied.

5.2.2 Soulé (Ph.D. Thesis)

The previously described work of Soulé et al. influenced the presentations in [SOU92a] where the previously gathered experiences were placed in an extended context. Here, Soulé examined the behavior of two parallel algorithms for parallel logic simulation based on the *THOR* logic simulator [ABC88a]. Both operate at the gate and register transfer level (RTL). The first algorithm is realized within a centralized time event-driven simulator, while the second algorithm is an implementation of the asynchronous conservative CMB-method.

Both simulators were first modeled on an ideal multiprocessor with uniform memory access for all processors. It was assumed that each reference to a memory location needs a constant amount of time regardless whether the access is performed locally or on a remote processor. This multiprocessor was realized as an emulation in a multiprocessor simulation environment.

Synchronous Simulator: The results obtained on the simulated multiprocessor with ideal load balancing and ideal memory access revealed that in the case of the synchronous algorithm different lengths for the evaluation of the units and the barrier synchronization due to the lock step processing mode present strong limits for speed-up. As an ideal load balance at no cost cannot be realized for true simulators, Soulé "corrected" this ideal view by introducing into his machine model step by step more realistic assumptions. In this way, he showed where the expected speed-up vanishes. In particular, the ideal load balance cannot be realized in general because of the varying evaluation times. Soulé reports theoretical acceleration factors between 10 and 17 on a 64 processor configuration.

To reflect the effects that occur on real systems, the same measurements were performed on an Encore Multimax using 16 processors. The speed-up for the synchronous algorithm yielded acceleration factors between 4 and 6. Another available multicomputer (DASH) only provided speed-ups between 2 and 3 using 16 processors.

The examination of the grain sizes of the event routines in terms of instructions per evaluation and the grain size of node updates revealed that on the one hand the duration of the unit evaluations does not diverge too much and is only a minor reason for poor load balancing. Large fanouts of nodes, on the other hand, cause a relatively high variance in the execution times of node updates and result in unequally balanced

load if the partitions are statically mapped and the behavior of the units cannot be predicted prior to simulation.

Asynchronous Simulator: The implementation of the parallel asynchronous simulator is based on the conservative CMB-approach with deadlock detection and resolution. The major obstacle to speed up the simulator was found to be the frequency of deadlocks. To improve the behavior of an initial version, the proposals from [SOB88a] were applied to prevent running into deadlocks that can be avoided by introducing implicit knowledge on the application into the algorithm. Furthermore, several variants are proposed by supplying additional information to the simulation routine such as using information from previous runs of the simulator, clocked registers, and grouping (*globbing*) of several units into higher functional units. Globbing might also increase the evaluation grain size and reduce the variance of the evaluation's duration.

In general, a speed-up of 16 to 32 could be achieved on the ideal multiprocessor. For the Encore, the acceleration factor lies between 6 and 9 for the improved versions.

Results: The comparison of both simulation strategies shows that the synchronous method is faster than the asynchronous CMB-algorithm for the examined circuits whose sizes range between 2,000 and 74,000 units. The overhead associated with the asynchronous version makes the theoretical advantages of the event-driven approach useless.

Another fact that should be mentioned is that the reported speed-ups are compared to the distributed algorithms that were running on one processor. This means that there is no comparison to a true sequential simulator that implements the same functionality but avoids the overhead that is associated with the distributed version even if it runs on a single processor. Hence, no general statements can be drawn from this experiments whether parallel simulation of the examined type can outperform a sequential simulator to obtain real acceleration.

The author concludes that the CMB-algorithm is badly suited to speed up parallel logic simulation. This is true at least at the gate-level or for higher abstractions although one expects for these layers larger evaluation grain sizes due to more complex units and therefore also a higher degree of parallelism. For a more detailed level such as switch-level simulation, further examinations have to be made. Soulé expects a better performance for the latter case because due to the larger number of units, the LPs are kept busy for longer periods of time.

5.2.3 Briner

In his Ph.D. thesis [BRI90a], Briner concentrated his work on Time Warp, a parallel asynchronous optimistic simulation scheme, originally described by Jefferson in [JEF85a]. He also considers critical path analysis (CPA) [BEJ85a]. This method is a scheme to calculate an upper bound on acceleration which is possible with a distributed simulator and must not be confused with critical path calculations from

VLSI design (see, e.g., [LDC91a]). The CPA uses a sequential simulator and calculates theoretical speed-up factors assuming either a limited or an unlimited number of processors for the distributed simulator. The CPA represents an alternative method to the measurements described in [WFC86a, BAS88a, BAI92a] to express parallelism in circuitry. Furthermore, Briner also incorporates the effects of the used memory access methods on multiprocessors into his reflections by comparing uniform versus nonuniform access for distributed and local memory.

Using the CPA metric for determination of potential parallelism, the author points out that there are several important aspects to be considered to get good speed-ups. First, the state saving in the Time Warp system has to be done efficiently. He proposes incremental state saving to avoid vast cemeteries of data that fill up memory. A second topic is the importance of partitioning with respect to the detailing degree of simulation. While partitioning might be a useful means in transistor-level simulation, it becomes crucial for the gate-level where the total evaluation grain size for each point in simulation time decreases by a magnitude of 10 which often results in poor load balance because processors run into idle phases.

By using several variants and optimizations for Time Warp, such as aggressive and lazy cancellation [FUJ90a] and statically sized bounding windows, measurements were performed which yielded speed-up factors of at most 25 with 32 processors on a shared memory architecture (BP-1000) compared to a sequential simulator. The average speed-up factor is about 6. The most promising approach from the variants seems to be a combination of lazy cancellation with bounding window where the size of the window strongly depends on the simulated circuit.

Another question in this context is raised by the fact that real systems only provide limited resources. For this reason, the partitions of the simulation model that are distributed onto the available processors generally contain more than one unit (*clustering*). Briner examined how to perform a rollback in such an environment. Should, in the case of a faulty speculative computation, all units of the processor be rolled back (*processor synchronization*) or must only the faulty effects on the single unit which is concerned by the the rollback be undone (*component synchronization*)? From his results, Briner concludes that combining component synchronization with lazy cancellation provides the best approach to speed up simulation for the five examined circuits.

These investigations are carried on in [BEK91a]. According to Briner, the circuit activity measurements described by Wong, Bailey and Soulé are a too restrictive approach to predict concurrency and parallelism in the case of asynchronous methods. Instead of their proposals, the CPA provides a much better metric for the attainable acceleration of parallel logic simulation in an asynchronous environment. Due to the different logical clocks, even events with different simulation time may be executed at the same moment in real time on different processors. Hence, the lock step forced by the synchronous strategies is removed allowing a much higher degree of concurrency. Another advantage of the asynchronous versions over the synchronous methods is that the LPs on different processors do not have to wait for the completion of all

other processors. Each LP can continue at its own speed without having to wait until all "concurrent" events are processed.

Briner proposes several improvements to additionally increase the speed-up of the basic Time Warp scheme. Among them are: incremental state saving, bounding window and different synchronization granularities.

Arguing from this approach, Briner points out the necessity of efficient partitioning algorithms to prevent disastrous effects of the needed synchronization among different LPs on the system's performance. High communication costs that are caused by a badly suited placement strategy may eat up the benefits which were obtained by using Time Warp.

5.2.4 Matsumoto and Taki

Matsumoto and Taki describe a parallel gate-level simulator based on Time Warp [MAT92a]. They obtained speed-ups of up to 50 with 64 processors compared to asynchronous conservative and synchronous methods on a distributed memory multiprocessor and argued from this point that Time Warp is superior to asynchronous conservative and to synchronous methods.

An interesting improvement is the proposal to send only one antimessage to the affected LPs during rollbacks. This message is the one which has the smallest timestamp of all antimessages that normally would have to be sent and which causes the cancellation of all messages sent by the same simulator with larger timestamps. Beside this, Matsumoto and Taki introduce some improvements in detail which yield better performance. However, their implementation depends on the underlying hardware and cannot be generalized.

The authors simulated four medium sized benchmarks from the ISCAS89 benchmark set [BBK89a] and recorded several supplementary statistics concerning rollback depth and frequency of rollbacks. For the larger of the simulated circuits they found that on the average rollback processing needed 5 % of the whole simulation time. As a metric for parallelism they used the average number of events evaluated per second of real time which resulted in the above mentioned speed-up. However, as Time Warp may process an event as soon as it is available, their measurement may contain events that are evaluated several times due to rollbacks at high speed. This may lead to false interpretations. The total number of events executed per second in a sequential version would be helpful to estimate the real speed-ups. Unfortunately, those numbers are not included in the article.

5.2.5 Karthik and Abraham

Karthik and Abraham examined switch level simulation on a network of workstations [KAA92a]. They used a conservative event-driven approach which was extended by a cycle-free partitioning scheme. The basic mapping units for partitions are strongly connected components that are extracted from the netlist graph to avoid feedback cycles over partition boundaries. The partitions of the model were ordered for propaga-

tion of event messages in a pipeline like fashion where the connections in the pipeline represent communication dependencies between partitions. The pipeline order assigns ranks to the partitions. Starting with the partitions containing the primary inputs at the lowest rank, all partitions with the same ranking level are simulated by the same processor. If there are more ranks than available processors the subcircuits are assigned in a manner that no cycles between processors occur. By that, Karthik and Abraham could avoid all deadlocks in their simulations.

Problems that might appear with this partitioning scheme are that the partitions can have very different sizes and become very large because of large feedback loops. To cope with low communication bandwidth, the authors propose to introduce buffers between the stages of the pipeline. Another proposal consists in compressing information by packing several signal values into one message.

Karthik and Abraham found an average speed-up factor of 2.3 with 3 processors for medium sized ISCAS89 benchmarks (10,000 to 33,000 transistors) with a maximum of 4.1 on 5 processors for one of the smaller circuits. They also discovered that their implementation provides similar speed-ups as a comparable simulator in a shared memory environment [MSA91a] so that one can hope that if communication effects and especially remote data accesses can be handled efficiently, there will not be too much differences in both approaches.

5.2.6 Manjikian and Loucks

Manjikian and Loucks [MAL93a] implemented a parallel gate-level simulator with unit-delay on a network of workstations. They used a hybrid approach for synchronization of the LPs. The single LPs are allowed to run in an optimistic way but event messages are only sent to other LPs when they are safe (i.e., they cannot be cancelled by stragglers). For this reason, the effects of rollbacks are kept local to the LPs.

Measurements with large circuits from the ISCAS89 benchmark suite yielded speed-up factors between 2 and 4.2 on 7 processors. The speed-up is calculated relative to a true sequential simulator. According to the authors, an important role is played by the applied partitioning scheme. They used cone partitioning [SMU87a] with enhancements to incorporate the estimated circuit behavior into the partitioning algorithm.

5.2.7 The DACAPO-III Project

The DACAPO-III system is a multi-level, mixed-mode simulation system for logic simulation [GRA90a]. It has been developed at the University of Dortmund and runs as a sequential and parallel version on a network of transputers. The parallel version uses Time Warp for synchronization of the LPs. The maximal number of processors that was used is 4. The statements in [GRA90a] concerning speed-up are unclear. It is only mentioned that speed-up strongly depends on the simulated model and no results from measurements are presented in this paper.

5.2.8 DISIM / Sang

Originally, DISIM was a sequential multi-level logic simulator developed by Daimler Benz AG in cooperation with the Technical University of Berlin. This simulator was extended by a parallel multi-level simulation (*PMLS*) component as part of the Esprit project 415. Synchronization is performed either by a conservative protocol or by Time Warp. PMLS was designed to run on general purpose multicomputers such as the Intel iPSC/2 hypercube [INT91a] or a network of transputers. This system supports all design levels. A prototype implementation was realized on the DOOM machine [BNO87a] in the object-oriented programming environment POOL [ANH90a]. Measurements showed that on the prototype with 100 processors speed-up factors in the range of 2 to 20 compared to the sequential version of DISIM could be achieved.

Another project that used sequential DISIM as basic simulator was the implementation of the parallel logic simulator *ParaDiSim* at the University of Dortmund ([SAN90a]) on a network of transputers. J. Sang and M. Sang realized three synchronization protocols: synchronous lock step, a simple asynchronous conservative method and Time Warp. While the first method brought no speed-up at all compared to the sequential version, the conservative approach led to acceleration of up to 3.4 with 16 processors. However, no linear relationship was found between acceleration and the number of processors used. The size of the benchmarks which were real designs ranged from 600 to 13,600 units. In the Time Warp implementation speed-up factors of about 1.6 on 4 transputers were observed for the smaller models and 1.5 for the largest one.

5.2.9 Bauer and Sporrer

At the Technical University of Munich, Bauer and Sporrer realized a parallel logic simulator based on Time Warp synchronization [BSK92a]. They used the sequential event-driven gate-level simulator *LDSIM* [KRA90a] as base for their work. The parallel version runs on a network of workstations. The authors propose incremental state saving for their simulator [BAS93a] to keep the overhead for memory administration low. The state saving is realized in an efficient way by modification of the event queue. Evaluated events are simply marked and left in the queue. When a rollback appears, the executed events are reactivated and no enqueue operations have to take place.

Bauer and Sporrer observed speed-up factors between 2 and 4 on 12 processors for medium sized ISCAS89 benchmarks in the range of 3,500 to 19,200 gates. The factors are given relative to the speed of the true sequential LDSIM simulator. Compared to the parallel algorithm on a single processor, the acceleration ranges from 4 to 6 on 12 processors for the observed circuits.

5.2.10 Luksch

In [LUW93a] and [LUK93a], Luksch reports the experiences from implementing a parallel version of LDSIM on the Intel iPSC/860 hypercube. He used a conservative algorithm and Time Warp for the synchronization of the LPs and enhanced the basic Time Warp algorithm by incremental state saving and lazy reevaluation.

Results for the conservative method are presented in [LUK93a] and speed-up factors of up to 2.8 were found for some circuits on 4 processors. For Time Warp, the obtained speed-up factors ranged between 1.2 and 4.7 for up to 16 processors with some medium sized circuits from the ISCAS89 benchmark suite. However, their measurements revealed no clear tendency between attainable speed-up and the number of used processors. Another important observation is the fact that for large circuits, the physical memory limitations cannot be neglected. Due to the fact that the logical clocks of the LPs may drift very far in Time Warp, there may be a huge amount of state saving information that has to be stored during simulation. Additional efforts to parallelize the code (notably the simulation support functions such as event queue management) instead of the model data yielded no promising results.

5.2.11 Lanchès

At the University of Stuttgart, Lanchès implemented a system for the evaluation of different strategies for parallel logic simulation on a network of transputers and also on a network of workstations to reflect the effects of different underlying hardware on simulation speed [LAB92a]. His system consists of a simulator that supports different synchronization strategies and several tools that provide a framework to examine monitoring data supplied by the simulators and to visualize them for evaluation and analysis. No results have been published so far, however.

5.2.12 Hoppe

Hoppe (Technical University of Berlin) simulated a parallel multi-level logic simulation system on a single processor to examine the influence of partitioning and dynamic load balancing on parallel logic simulation [HOP91a]. The LPs were synchronized either by Time Warp or with a conservative method. Hoppe measured speed-up factors between 2 and 16 on up to 64 virtual processors with the conservative method and acceleration rates of up to 17.5 when Time Warp was used. The initial partitioning strategy tries to minimize the costs for communication and also respects given partition sizes which were calculated from estimated activities of the units.

Dynamic load balancing is realized on the base of estimated progress of the local virtual time (LVT). When load imbalance appears, small portions of the circuit are moved from the LP with the largest progress to the LP with the smallest progress in LVT. As the whole system is simulated on a single processor, this can be done easily by a central instance. Hoppe reports that the simulations yielded an efficiency

of 34% to 81% of the maximal possible speed-up with some small benchmarks. With dynamic repartitioning the speed-up could be increased between 2% and 38%.

5.2.13 Simic and Ortner

Simic and Ortner implemented the parallel multi-level and multi-mode logic simulator DELSIM on a network of transputers at the Technical University of Berlin [SIO93a]. They used Time Warp for synchronization of the logical processes. A basic dynamic repartitioning scheme is included in their system. A central controller records the LVTs and initiates load balance actions if a threshold is passed. The authors report speed-up factors of up to 3.3 on 6 processors with a small circuit without load balancing and further improvement factors between 1.1 and 2.1 for dynamic load balancing.

5.3 Parallelization of VHDL Simulators

5.3.1 VHDL (VHSIC Hardware Description Language)

VHDL provides a means to describe a circuit under design in a high-level language at very different detailing levels [VHD87a]. Its abilities range from description on the system layer and register transfer level down to the design at transistor and switch level. VHDL supports hierarchical designs and facilitates the development and design process of new circuits. The user may specify the circuit through one of three different views. The structural view describes the interconnections and dependencies between the different components. The behavioral view represents the dynamic behavior of the cell under design and the logical view or data flow view may be used to transform boolean equations into a circuit description. VHDL provides flow control statements, sequential statements, and statements for concurrent signal and variable assignments. Additionally, VHDL allows nested block structures. A special block type is the *process* statement that is often used for behavioral description of a circuit as a block of sequential statements which is cycled through in an endless loop. The execution of a process block is influenced by wait conditions on signal changes (wait statement). Beside these features, VHDL defines basic primitives such as simple gates, and it supports different types of logic values and I/O operations which are supplied through libraries (packages).

VHDL descriptions are used for validation of circuits through simulation, as the base for the further steps within the design process, and for documentation of the target circuits. For simulation of the models, usually each VHDL model is compiled into a custom-mode simulator that allows to verify the behavior of the circuit.

5.3.2 Vellandi and Lightner

Vellandi and Lightner parallelized a VHDL simulator by restructuring the VHDL description of the simulated models and modifying the compiling of the model de-

scription [VEL90a, VEL92a]. The concurrent statements were used in an ad hoc manner as parallel components. For the other statement types, techniques for language parallelization using graph rewriting are applied.

A second proposal to accelerate simulation consists in using efficient techniques to realize parallel versions of the simulation support functions (evaluation, event queueing). The authors simulated the modified VHDL-descriptions on a Connection Machine (CM2) with additional support to make the SIMD machine behave almost like a MIMD architecture via an emulation layer. They obtained speed-ups in the range of 1.4 to 10 on the CM-2 for quite small circuits by mapping each simulated unit onto a dedicated processor. They also mention that by restructuring the simulation support functions a much higher degree of parallelism can be obtained than outlined in [SOB88a, WFC86a, BAS88a].

5.3.3 Wilsey, Palaniswamy, Chawla, and Aji

Another parallel simulator based on VHDL has been built by Chawla and Wilsey at the University of Cincinnati [CHW91a]. They implemented an event-driven asynchronous version using the Time Warp strategy with various modifications. As VHDL provides a so-called *delta-delay* timing to model causal dependencies between events with the same simulation times, the notion of timestamps has to be extended to a pair

```
(<simulation time>, <delta-cycle number within actual timestep>).
```

The comparison of two timestamps is performed in lexicographical order from left to right. The delta delay is an infinitesimal time that does not advance the *normal* simulation time but is used to distinguish between different cycles within VHDL constructs. The delta time is set to zero for the first events of a timestep. If the evaluation of an event creates new events with the same simulation time, the delta value of the new events is incremented by one. If new events are scheduled for a future simulation time, the delta value is reset to zero for these events. Similar constructs for handling events with the same simulation time are described in [MEH91c].

In [WIP92a] an extension to the simulator is introduced, called *rollback relaxation*. VHDL process blocks can be classified into two groups: stateless processes and processes with states due to previous iterations of the process loop. The authors provide means to relax the processing of stateless processes during a rollback resulting in less processing overhead because no state saving and checkpointing has to be performed.

In general, concurrent statements and processes behave in different ways during the simulation. To be able to proceed in a uniform way, all concurrent statements and blocks are transformed into processes before the relaxation algorithm is started. In a second stage, all stateless processes are determined and marked for relaxation of rollbacks.

It was observed that an average of 74 % of all processes qualify for the proposed

improvement and that speed-up factors of 1.12 to 1.46 could be obtained by introduction of relaxation. These results were obtained from tests with four small circuits. As no information is provided on the exact circuit sizes and on the number of processors for which these results were achieved, it remains questionable whether the observations can be generalized to larger designs.

Another modification to the basic Time Warp algorithm is proposed in [PAW92a]. The reevaluation of events after rolling back an LP often leads to the same state as if the straggler or antimessage had not appeared. Hence there exists a potential to avoid unnecessary work by just retaining the old state and continue as if no error had occurred. An efficient technique to compare states is lined out which consists in a modification of the event routine's code by introducing flags which indicate a possible difference in the states. If the reevaluation qualifies for a possible change, the states have to be recalculated to reflect the changed conditions. Otherwise, the LP can continue with the state immediately before the rollback because a reevaluation would produce exactly the same state. By that, the rollback algorithm can use a shortcut and skip over the reevaluation phase. This proposal represents an implementation of *lazy reevaluation* as described in [FUJ90a].

Measurements for this technique were performed with three of the small benchmarks mentioned before comparing execution times and number of rollbacks to the results provided by a traditional aggressive cancellation version of Time Warp. The new algorithm performed better than the aggressive cancellation method, but from the reported results it should be considered as a minor optimization only.

In [APW93a], the effects of combining different improvements for Time Warp are examined: bounded time window, rollback relaxation, lazy cancellation and periodic checkpointing. The work reveals that a simple combination of all proposals does not always improve performance of parallel logic simulation. The influence of the different modifications also depends on the simulated model. The authors conclude that the combination of bounded time window, lazy cancellation, and periodic checkpointing provided the most general approach for optimization of Time Warp.

5.3.4 The Simulator Coupling Project

In cooperation with several industrial partners, the Simulator Coupling Project tries to combine existing parallel logic simulators that may simulate different levels of the design hierarchy into one multi-level and multi-mode simulator [BLN90a]. A subset of VHDL was chosen as the hardware description language for the circuits to be simulated.

The synchronization of the different simulators is based on Time Warp with lazy cancellation. The circuit components are examined at compile time for suitability for simulation by the incorporated simulators. At the start of the simulation, the circuit components are distributed to the corresponding simulators and a coordination and data exchange layer provides communication between the simulators. Time Warp is realized within this layer.

Furthermore, a dynamic adaptive control behavior is planned where a coordinator may dynamically modify the synchronization strategy ranging from lock step simulation to totally asynchronous simulation. Results are not presented so far for this project.

5.3.5 The DVSIM-Project

As part of a project to evaluate general principles of parallel and distributed simulation at the University of Saarland, we have parallelized an existing VHDL simulator (vsim, University of Pittsburgh) [LEV93a] using asynchronous event-driven strategies. Recently a conservative strategy was implemented using the Chandy-Misra deadlock detection and recovery scheme [CHM81a]. The system is running on an Intel iPSC/860 hypercube and on a network of workstations. Both versions are based on the MMK system [BEL90a] providing an unique interface for programming in both environments.

We use four different partitioning schemes: Cycle-free partitioning, Round Robin, Kernighan-Lin [KEL70a] and SOCCER [RUN88a]. A simulated annealing based algorithm will be integrated soon. In addition to the results gained from the usual experiments, the parallelism available from our benchmarks [BBK89a] is also examined using the critical path approach [BEJ85a] and oracle lag [SWF87a] to reflect both the ideal speed-up and the effects of real communication.

Furthermore, we are implementing several variants of Time Warp to be able to faithfully compare conservative and optimistic approaches. To ease the design of new synchronization strategies, all protocol dependent actions are encapsulated within three procedures of the distributed simulator following Reynolds' and Dickens' proposal described in [RED89a].

In order to obtain detailed insights into the behavior of the parallel simulation algorithms, monitoring mechanisms and statistical evaluation tools [STU93a] have been realized. A variant of the ParaGraph system [HEF91a] is used for graphical animation purposes. First measurements show that for some benchmark models speed-up is possible with our DVSIM system even for conservative strategies, but it is yet too early to draw general conclusions from these experiments.

6 Related Topics: Partitioning of Circuits, Load Balance and Model Granularity

As the last section revealed, the basic algorithms for distributed simulation often show poor performance. This is not only a problem for distributed simulation but also for many other parallel applications. There is a class of closely related aspects that influence the performance of such applications. The dependencies among them are rather involved and the known partial solutions represent a vast space of combinations for experimentation. In this section, the basic factors that influence performance and

which are relevant to parallel logic simulation are sketched.

6.1 Partitioning and Mapping

In this section we will describe different approaches to do data partitioning of logic circuits for parallel simulation. The partitioning and mapping problem is caused by the fact that bounded resources as well as communication requirements in a distributed environment have to be respected. An optimal mapping is often hard to find due to the NP-completeness of the problem and heuristics must be applied. Fundamental aspects of keeping a simulation balanced will also be mentioned.

6.1.1 Random Partitioning

The simplest method of partitioning is just to scatter the units randomly over the available processors. This might be a bad solution at first sight but it provides at least a lower bound for the comparison of the performance and behavior of other placement strategies. In [FRA86a], Frank even argues that random partitioning will perform as well as anything else if the processors can always be kept busy, an assumption that hardly holds for parallel logic simulation.

There exist several slight modifications of this principle. One of them is mentioned in [SOU92a] as a kind of round robin mapping assigning units as well as the connected output nodes in a modulo-like fashion to the processors. Another enhancement is to assign a unit to a processor that has the lowest current load (for example, measured in the number of transistor equivalents) to achieve a nearly equal distribution of work [KRA88a].

The random algorithms usually do not respect any application specific knowledge but cause low computational costs and are very fast. In the following sections additional information on logic simulation will be used in order to improve the mapping of the circuits to partitions.

6.1.2 String Partitioning

The string partitioning method ([SOU92a, BRI90a, MAT92a]) first distributes the input nodes and registers in a random way onto the available processors because these units are supposed to create many new events during the whole simulation. For this reason they will produce heavy load and should be separated by assigning them to different partitions. Starting with the input nodes and the outputs of the registers the netlist is scanned for units driven by those nodes. The first unit that fulfills this requirement is added to the corresponding partition. The algorithm continues in a depth-first scheme by following always only one out of several fanout branches until either an external output of the circuit or an already assigned unit is encountered. If there remain unassigned units, one of them is randomly chosen, placed into the partition with the least number of units and the depth-first search is repeated for this unit. This algorithm is repeated until all units belong to a partition [AGR86a].

In most cases this method distributes units that are driven by the same signals, in particular from nodes with high fanout, into different partitions so that they are possibly evaluated in parallel during the simulation when a change on the driving signal occurs.

6.1.3 Notion of Regions

Several units that are closely connected (e.g., transistors sharing sources or drains) form so-called *regions*. Because of their strong coherence and in order to minimize communication costs complete regions should be mapped onto single processors.

In the *packed* partitioning algorithm [KRA88a] the regions are ordered according to the number of their units. Starting with the largest region, the regions are placed on the processor with the smallest number of assigned units. For this mapping an equally loaded distribution may be attained so that all processors are assigned approximately the same number of units.

A similar approach is proposed by Karthik and Abraham [KAA92a]. As an extension to the previous method, they provide cycle-free partitioning by requiring that feedback loops must be included in one single region and by assigning only complete regions to processors.

In [BAS93b], Bauer and Sporrer propose another partitioning algorithm based on regions. First, the strongly connected regions are calculated (petals). The petals are combined to so-called corollas in a second step. The criterion for combination of the regions is to minimize the connectivity between different corollas in order to minimize the effects of message passing during simulation. Additionally, a maximum size for the corollas may be specified.

6.1.4 Minimizing Communication Costs

In a distributed-memory system the underlying communication network plays an important role. The relation between computational and communication overhead must be balanced if one wants to avoid that high communication costs render the whole approach of distributed programming worthless.

The problem of minimizing communication costs is essentially a graph partitioning problem where the units represent the vertices of the graph and the connections between them are the edges.

The following scheme tries to minimize communication costs on an empirical base. The mapping algorithm takes into account the effects of communication. The parameters for assigning costs to communication connections are usually taken from previous runs by accumulating statistics for the communication channels in the circuit. They are needed to calculate the impact of the different signal propagation frequencies onto the communication overhead. The algorithm tries to minimize communication across partition boundaries by keeping units connected by signals with heavy duty communication local to a single partition.

Basic algorithms for this approach were developed by Kernighan and Lin and are described in [KEL70a]. Briner also proposed to include information on signal activities into the calculation of partitions and uses a recursive bisection method to obtain more than two partitions [BRI90a].

In the case of a shared-memory environment the same methods should also work but some modifications are required. The problem is that one cannot exactly predict on which processor the data structures that represent the units of the circuits are located because of the desirable transparency of data allocation. For this reason the communication costs should be replaced by data access costs to partition the circuits.

6.1.5 Bitslice and Hierarchical Partitioning

Methods using bitslice [SOU92a] and hierarchical partitioning [ARN85a] strongly depend on the simulated model and the availability of the necessary information. Bitslice mapping is based on the observation that on all bits of a word (e.g., a word representing a bus) the same operations are performed simultaneously. If the simulation of each bitslice is mostly independent from the other bitslices, it can be carried out concurrently and each bitslice may be placed entirely in one partition.

The hierarchical approach is based on the assumption that in a hierarchical design the units at the same layer are usually independent units and can be viewed as a basic elements for partitioning.

The problem with these methods is whether there is enough inherent parallelism within such coarse grained divisions. In particular for the hierarchical designs often only one component out of several might be active at a time.

6.1.6 Cone Partitioning

Cone Partitioning [SMU87a] is well suited for clocked circuits that contain latches. Starting with the latches of a circuit, all units that feed a latch are added to the input cone of that latch. This procedure is repeated recursively for the added units until either a primary input or the output of a latch is reached. As cones may overlap, several units have to be replicated if they are placed in different partitions. The assignment of cones to partitions may be done in different ways. The simplest one is random mapping. Manjikian and Loucks [MAL93a] propose to collect data on circuit activity from previous or partial simulation runs and to use the data as weighting factors for the units. The weight of cones is determined by the weights of its units. In an iterative improvement technique which tries to balance the load of the partitions and to minimize the number of replications, the cones are assigned to partitions.

6.1.7 Simulated Annealing

Simulated annealing is a method that is often used in placement during later steps of the chip design process. This placement strategy has been applied for parallel logic

simulation by Frank in [FRA85a]. As it is a general method, it will not be detailed here. For the basic principles, the reader is referred to [KGV82a, SES85a].

6.2 Dynamic Partitioning — Load Balance

Dynamic partitioning schemes for parallel logic simulation have been reported in [KRA88a, SAN90a, LUK93a]. Kravitz and Ackland state that dynamically repartitioning of a model during a running simulation will not provide significant improvements because the overhead created by load determination, data exchange for the unit migration, and bookkeeping on the actual locations of the units would overcome the benefits from having equal loads on all processors.

J. Sang and M. Sang implemented a basic facility for dynamic load balancing [SAN90a]. They used a weighted ratio of the number of evaluations per unit and the number of created events by a unit as a base for repartitioning decisions. The repartitioning is performed through a central control instance which monitors the load data of the processing nodes. If the LPs' loads differ too much, the simulation is stopped, the partitioning algorithm is restarted, and the units are collected and redistributed onto the processors. The results are already promising for this simple version of dynamic load balancing. The speed-up that was obtained with the distributed version of DISIM was still increased by dynamic load balancing.

Luksch and Weitlich also chose dynamic partitioning as a means to speed up parallel logic simulation [LUW93a]. They defined load as the difference between the local clocks of the LPs. To reestablish load balancing, one or several units are moved from the LP with the lowest local clock to the LP which has the maximum simulation time. For the choice of units, the authors give several criterions: the complexity of evaluation of a unit, the activity of the unit, and the effect of the unit on communication in the system. They also point out that the costs for transmission of state histories in Time Warp must not be neglected. Measurements have not yet been performed. Similar approaches for dynamic load balancing based on the local virtual time of the LPs have been pursued by Hoppe and by Simic and Ortner in [HOP91a] and [SIO93a], respectively.

In parallel logic simulation load balancing principles may also be realized through other methods. Soulé proposes the implementation of a distributed event queue from which all idle processors can request additional work if they complete while other processors are still running [SOU92a]. The same principle should be possible for the work of the update phase of nodes. Soulé realized this method for a centralized time algorithm on a shared memory machine. These schemes are well suited for synchronous lock step simulation with centralized time or hardware simulators and simulation machines. For asynchronous simulators where each LP runs a subset of the whole model they do not seem to fit well.

6.3 Computation Versus Distribution Overhead

As mentioned in Section 5, the granularity of the models is an important factor for the parallelism and concurrency that can be exploited by a distributed and parallel application like parallel logic simulation.

Usually, the number of events which are simulated in parallel logic simulation decreases if the simulation is performed at higher abstraction levels. This fact was confirmed by measurements conducted by Soulé and Blank in [SOB88a]. On the other hand, the cpu time needed to simulate one event increases for the higher design layers. Interestingly, in measurements of the total time of a simulation run, Soulé and Blank found that each step towards higher abstraction results in gaining speed-up of approximately 10. The higher the abstraction level of the simulated circuits, the faster they can be simulated. This is mainly caused by the fact that each stage of abstraction has to be paid by a loss of accuracy of the simulated model and in a reduced number of units at the higher levels.

Often, however, very detailed simulations are necessary to obtain the high accuracy needed to verify the behavior of the unit under design. The large run times for these simulations can only be reduced if the acceleration efforts through parallelization do not introduce too much overhead, and if the time spent in the additional work needed for the distributed application can be compensated by the benefits of executing parts of the application in parallel.

A main source of overhead are the communication costs between the distributed components to propagate signal changes and to synchronize the components. In a shared memory environment, the additional costs are due to synchronization by locks for data structures or similar mechanisms such as semaphores. The costs of these operations must be kept low. If the used hardware has large latencies for these operations and not much work is supplied to the single processors, they will spend most of their time in waiting for the arrival of messages or until a lock is available. Hence, it is desirable to provide enough work to each processor to keep it busy. This leads to the conclusion that simulation of large circuits and simulation of circuits where units have large evaluation times for the calculation of a new state are best suited for parallel logic simulation.

Another problem is whether several units should be simulated by a single LP or whether each unit is simulated by a dedicated LP. Even with unlimited resources, the second approach to simulate each small unit by a dedicated simulator is doomed to failure right from the beginning because the synchronization and administration overhead is much too large. Also, different granularities of the units in a multi-level design would slow down a parallel simulation because due to differing evaluation times of the units the simulators would be blocked most of the time waiting for externally caused events.

A better approach consists in forming clusters of units that are simulated by one LP. As only one process can be active at a given time on one processor, only one simulator (LP) should be placed on each processor. The assignment of units to

clusters is solved by an appropriate partitioning algorithm (see Section 6.1) where estimates of the activity of units and the duration of the event evaluations may be applied if they are available. This helps to balance the load of all processors. One disadvantage of this *single cluster* approach is that the whole cluster may block if events for only one unit are pending due to external dependencies. This, however, can be avoided in many cases by the use of an appropriate partitioning algorithm.

These two methods are the extreme cases. For practical implementation, both schemes may be combined, for example, by using multithreading for running several LPs on a single processor.

7 Conclusions

Much work has been done on parallel logic simulation. Several approaches seem to be quite promising for significantly speeding up simulation times. Among them are hardware accelerators that provide the ability to simulate restricted models of a given type at a very high speed. For general use, however, they are usually too expensive and too inflexible to be of interest.

For software simulation of logic circuits, two main proposals have been made: synchronous centralized time and asynchronous simulation. Although the first approach allows a simple implementation, usually not enough parallelism is available within the given models so that the distributed simulators often run slower than good sequential tools. Asynchronous simulation can provide a higher degree of parallelism by removing the restriction that all simulator components must advance their simulation time with the same speed. However, even for the asynchronous schemes, speed-up is hard to obtain. The most interesting approach among all known synchronization methods seems to be Time Warp which has maximally decoupled simulator components. Recent work has concentrated mainly on this approach.

However, until now no conceptual work has been done covering all aspects of asynchronous logic simulation in their entirety including experimental comparison of different synchronization strategies on general purpose machines for an accepted set of benchmarks with reasonably sized circuits. Also, different partitioning strategies should be examined for their applicability to parallel logic simulation. Finally, it would be very interesting to investigate whether general statements on the suitability of a given circuit for parallelization based on its characteristics can be made. This last point might have major impact on parallelization efforts because then the best suited simulator for a given model could be chosen from data gathered during the design process.

Acknowledgements
The author wants to thank Friedemann Mattern as well as some of the author's colleagues for giving useful comments on earlier versions of this paper.

References

[ABC88a] Alverson, B., Blank, T., Choi, K., Hwang, S., Salz, A., Soulé, L., and Rokicki, T. (1988) *THOR User's Manual: Tutorial and Commands*, TR CSL-TR-88-348, Stanford University

[AGR86a] Agrawal, P. (1986) *Concurrency and Communication in Hardware Simulators*, IEEE Trans. on Computer-Aided Design, vol. 5, pp. 617 – 623

[ANH90a] Annot, J. and den Haan, P. (1990) *POOL and DOOM: The Object Oriented Approach*, Treleaven, P.C. (ed.), Wiley

[ARN85a] Arnold, J. (1985) *Parallel Simulation of Digital Circuits*, Master's thesis, MIT

[APW93a] Aji, S., Palaniswamy, A., and Wilsey, P. (1993) *Interactions of Optimizations to a Time Warp Synchronized Digital System Simulator*, Proc. European Simulation Multiconference, pp. 593 – 597

[BAI92a] Bailey, M. (1992) *How Circuit Size Affects Parallelism*, IEEE Trans. on Computer-Aided Design, vol. 11, pp. 208 – 215

[BAS88a] Bailey, M. and Snyder, L. (1988) *An Empirical Study of On-chip Parallelism*, Proc. 25th Design Automation Conference, ACM/IEEE, pp. 160 – 165

[BAS93a] Bauer, H. and Sporrer, C. (1993) *Reducing Rollback Overhead in Time Warp Based Distributed Simulation with Optimized Incremental State Saving*, Proc. 26th Annual Simulation Symposium ASS 93, Washington

[BAS93b] Sporrer, C. and Bauer, H. (1993) *Corolla Partitioning for Distributed Logic Simulation of VLSI-Circuits*, Proc. 7th Workshop on Parallel and Distributed Simulation, vol. 23, no. 1, pp. 85 – 92

[BBB87a] Bryant, R., Beatty, D., Brace, K., Cho, K., and Sheffler, T. (1987) *COSMOS: A Compiled Simulator for MOS Digital Circuits*, Proc. 24th Design Automation Conference, ACM/IEEE, pp. 9 – 16

[BBK89a] Brglez, F., Bryan, D., and Kozminski, K. (1989) *Combinational Profiles of Sequential Circuits*, In Proc. ISCAS '89

[BEJ85a] Berry, O. and Jefferson, D. (1985) *Critical Path Analysis of Distributed Simulation*, Proc. Distributed Simulation Conference, San Diego, pp. 57 – 60

[BEK91a] Briner, Jr., J., Ellis, J., and Kedem, G. (1991) *Breaking the Barrier of Parallel Simulation of Digital Systems*, Proc. 28th Design Automation Conference, ACM/IEEE, pp. 223 – 226

[BEL90a] Bemmerl, T. and Ludwig, T. (1990) *MMK – A Distributed Operation System Kernel with Integrated Loadbalancing*, Proc. CONPAR 90, VAPP IV, LNCS:457, pp. 744 – 755

[BLN90a] Bechthold, M., Leyendecker, T., Niemeyer, M., Oczko, A., and Oczko, C. (1990) *Das Simulatorkoppelungsprojekt*

[BNO87a] Bronnenberg, W., Nijman, L., Odijk, E., and Twist, R. (1987) *DOOM: A Decentralized Object Oriented Machine*, IEEE Micro, vol. 10, pp. 52 – 69

[BRI90a] Briner, Jr., J. (1990) *Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time*, PhD thesis, Duke University, Durham, NC, TR90-38

[BRY77a] Bryant, R. (1977) *Simulation of Packet Communication Architecture Computer Systems*, TR-188, LCS, MIT

[BSK92a] Bauer, H., Sporrer, C., and Krodel, T.H. (1992) *On Distributed Simulation Using Time Warp*, Technical University of Munich, TR TUM-I9212, SFB report 342/9/92A

[CHA85a] Chamberlain, R. (1985) *Lsim: A Gate-Switch Level Logic Simulator*, Master's thesis, Department of Computer Science, Washington University, St. Louis

[CHM81a] Chandy, K. and Misra, J. (1981) *Asynchronous Distributed Simulation Via a Sequence of Parallel Computations*, Comm. of the ACM, vol. 24, pp. 198 – 206

[CHW91a] Chawla, P. and Wilsey, P. (1991) *Synchronizing Distributed VHDL Simulation*, TR 131-4-91-ECE, University of Cincinnati

[FRA85a] Frank, E. (1985) *A Data-driven Multiprocessor for Switch-level Simulation of VLSI Circuits*, PhD thesis, Carnegie-Mellon University

[FRA86a] Frank, E. (1986) *Exploiting Parallelism in a Switch Level Simulation Machine*, Proc. 23rd Design Automation Conference, ACM/IEEE, pp. 20 – 25

[FUJ90a] Fujimoto, R. (1990) *Parallel Discrete Event Simulation*, Comm. of the ACM, vol. 33, pp. 30 – 53

[GRA90a] GRABIENSKI, P. (1990) *DACAPO-III: Parallel Multilevel Hardware Simulation on Transputers*, Proc. 2nd International Conference on Applications of Transputers, pp. 559 – 564

[HAF85a] HAHN, W. and FISCHER, K. (1985) *An Event-Flow Computer for Fast Simulation of Digital Systems*, Proc. 22nd Design Automation Conference, ACM/IEEE, pp. 338 – 344

[HEF91a] HEATH, M.T. and ETHERIDGE, J. (1991) *Visualizing Performance of Parallel Programms*, TR ORNL/TM-11813, Oak Ridge Nat. Lab.

[HIL85a] *HILO-3 User's Manual* (1985) Fareham, England

[HOP91a] HOPPE, F. (1991) *Ein Verfahren zur Synchronisation und Lastverteilung für die parallele Mehrebenenlogiksimulation*, PhD thesis, Department of Computer Science, Technical University of Berlin

[HWB84a] HWANG, K. and BRIGGS, F.A. (1984) *Computer Architectures and Parallel Processing*, McGraw–Hill

[INT91a] *Intel iPSC/2 and iPSC/860 User's Guide* (1991), Intel Cooperation

[JEF85a] JEFFERSON, D. (1985) *Virtual Time*, ACM Transactions on Programming Languages and Systems, vol. 7, pp. 404 – 425

[JES85a] JEFFERSON, D. and SOWIZRAL, H. (1985) *Fast Concurrent Simulation Using the Time Warp Mechanism*, Proc. Conference on Distributed Simulation, pp. 63–69

[KAA92a] KARTHIK, S. and ABRAHAM, J. (1992) *Distributed VLSI Simulation on a Network of Workstations*, Int. Conf. on Computer Design: VLSI in Computers & Processors, pp. 508 – 511

[KEL70a] KERNIGHAN, B. and LIN, S. (1970) *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell System Technical Journal, vol. 49, pp. 291 – 307

[KGV82a] KIRKPATRICK, S., GELATT, C., and VECCHI, M. (1982) *Optimization by Simulated Annealing*, Technical report, Watson Res. Center, Yorktown Heights, IBM

[KRA88a] KRAVITZ, S. and ACKLAND, B. (1988) *Static vs. Dynamic Partitioning of Circuits for MOS Timing Simulator on a Message-based Multiprocessor*, In UNGER, B. and JEFFERSON, D. (eds.), Proc. Distributed Simulation Conference, San Diego, pp. 136 – 140

[KRA90a] KRODEL, T. and ANTREICH, K. (1990) *An Accurate Model for Ambiguity Delay Simulation*, Proc. EDAC, pp. 122 – 127

[LAB92a] Lanchès, P. and Baitinger, U. (1992) *A Parallel Evaluation Environment for Distributed Logic Simulation*, In Stephenson, J. (ed.), Modelling and Simulation 1992, pp. 465 – 469

[LDC91a] Liu, L.-R., Du, D., and Chen, H.-C. (1991) *An Efficient Parallel Critical Path Algorithm*, Proc. 28th Design Automation Conference, ACM/IEEE, pp. 535 – 540

[LEV93a] Levitan, S. (1993) *VCOMP & VSIM Reference Manual, Edition 1.2.1*, Department of Electrical Engineering, University of Pittsburgh

[LLB89a] Lin, Y., Lazowska, E., and Bailey, M. (1989) *Comparing Synchronization Protocols for Parallel Logic-Level Simulation*, TR89-09-06, Department of Computer Science and Engineering., University of Washington, Seattle, Washington

[LUK93a] Luksch, P. (1993) *Evaluation of Three Approaches to Parallel Logic Simulation on a Distributed Memory Multiprocessor*, Proc. 26th Annual Simulation Symposium, Arlington, pp. 2 – 11

[LUW93a] Luksch, P. and Weitlich, H. (1993) *Time Warp Parallel Logic Simulation on a Distributed Memory Multiprocessor*, Proc. SCS European Simulation Conference, Lyon, pp. 585 – 589

[MAL93a] Manjikian, N. and Loucks, W. (1993) *High Performance Parallel Logic Simulation on a Network of Workstations*, Proc. 7th Workshop on Parallel and Distributed Simulation (PADS), vol. 23, no. 1, pp. 76 – 84

[MAT92a] Matsumoto, Y. and Taki, K. (1992) *Parallel Logic Simulation on a Distributed Memory Machine*, Proc. European Conference on Design Automation, pp. 76 – 80

[MEH91c] Mehl, H. (1991) *Breaking Ties Deterministically in Distributed Simulation Schemes*, TR SFB124-217/91, Department of Computer Science, University of Kaiserslautern

[JOM89a] Jöhnk, R. and Marwedel, P. (1989) *Mimola Reference Manual, Ver. 3.45*, TR 8902, Department of Computer Science, University of Kiel

[MIS86a] Misra, J. (1986) *Distributed Discrete-Event Simulation*, Computing Surveys, vol. 18, pp. 39 – 65

[MSA91a] Müller-Thuns, R., Saab, D., and Abraham, J. (1991) *Parallel Switch-Level Simulation for VLSI*, Proc. EDAC, pp. 324 – 328

[NAG75a] Nagel, L. (1975) *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, University of California, Berkeley, TR UCB ERL-M250, Electronics Research Lab.

[PAW92a] Palaniswamy, A., Aji, S., and Wilsey, P. (1992) *An Efficient Implementation of Lazy Reevaluation*, Proc. 25th Annual Simulation Symposium

[PFI82a] Pfister, G. (1982) *The Yorktown Simulation Engine: Introduction*, Proc. 19th Design Automation Conference, ACM/IEEE, pp. 51 – 54

[RED89a] Reynolds, P. and Dickens, P. (1989) *SPECTRUM: A Parallel Simulation Testbed*, Hypercube Conference, March 1989

[REY88a] Reynolds, P. (1988) *A Spectrum of Options for Parallel Simulation*, Proc. Winter Simulation Conference, pp. 325–332

[ROE93a] Rössig, K. (1993) *Zum Stand der Forschung auf dem Gebiet der parallelen Logiksimulation*, TR AIS-9, Department of Computer Science, University of Oldenburg

[RUN88a] Runo, D. (1988) *Das Graphpartitionierungsproblem und seine Lösungsmethoden*, Master's thesis, GMD, Bonn

[SAN90a] Sang, J. and Sang, M. (1990) *Untersuchung von Algorithmen zur verteilten ereignisgesteuerten Simulation am Beispiel eines Multi-Level-Simulators auf Transputerbasis*, Master's thesis, University of Dortmund

[SES85a] Sechen, C. and Sangiovanni-Vincentelli, A. (1985) *The TimberWolf Placement and Routing Package*, IEEE Journal of Solid-State Circuits, vol. 20, pp. 510 – 522

[SIO93a] Simic, N. and Ortner, H. (1993) *Partitioning Strategies Within a Distributed Multilevel Logic Simulator Including Dynamic Repartitioning*, submitted to Proc. EDAC

[SMU87a] Smith, S., Mercer, M., and Underwood, B. (1987) *An Analysis of Several Approaches to Circuit Partitioning for Parallel Logic Simulation*, Proc. Int. Conference on Computer Design, IEEE, pp. 664 – 667

[SOB87a] Soulé, L. and Blank, T. (1987) *Statistics for Parallelism and Abstraction in Digital Simulation*, Proc. 24th Design Automation Conference, ACM/IEEE, pp. 588 – 591

[SOB88a] Soulé, L. and Blank, T. (1988) *Parallel Logic Simulation on General Purpose Machines*, Proc. 25th Design Automation Conference, ACM/IEEE, pp. 166 – 171

[SOG89a] Soulé, L. and Gupta, A. (1989) *Analysis of Parallelism and Deadlocks in Distributed-Time Logic Simulation*, TR CSL-TR-89-378, Stanford University

[SOU92a] SOULÉ, L. (1992) *Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms*, PhD thesis, TR CSL-TR-92-527, Stanford University

[STU93a] STURM, B. (1993) *YES - Ein Werkzeug zur Analyse verteilter Simulationsalgorithmen*, Master's thesis, Department of Computer Science, University of Kaiserslautern

[SWF87a] SWOPE, S. and FUJIMOTO, R. (1987) *Optimal Performance of Distributed Simulation Programs*, Proc. Winter Simulation Conference, pp. 612–617

[TER83a] TERMAN, C. (1983) *Simulation Tools for Digital LSI Design*, PhD thesis, MIT

[VEL90a] VELLANDI, B. (1990) *Parallelism Extraction and Program Restructuring for Parallel Simulation of Digital Systems*, PhD thesis, University of Colorado

[VEL92a] VELLANDI, B. and LIGHTNER, M. (1992) *Parallelism Extraction and Program Restructuring of VHDL for Parallel Simulation*, submitted to Proc. EDAC

[VHD87a] *IEEE Standard VHDL Language Reference Manual* (1987), IEEE Std. 1076–1987, edition 1987

[WFC86a] WONG, K., FRANKLIN, M., CHAMBERLAIN, R., and SHING, B. (1986) *Statistics on Logic Simulation*, Proc. 23rd Design Automation Conference, ACM/IEEE, pp. 13 – 19

[WIP92a] WILSEY, P. and PALANISWAMY, A. (1992) *Rollback Relaxation*, TR 135-2-92-ECE, University of Cincinnati

[ZIM80a] ZIMMERMANN, G. (1980) *MDS – The Mimola Design Method*, Journal of Digital Systems, vol. 4, pp. 337 – 369

Contents