Multiprocessor implementation of an event-driven circuit simulator

Evaluation of design options

Luís Miguel Silveira

INESC/IST

R. Alves Redol, 9

1000 Lisboa

PORTUGAL

Abstract

This report sumarizes and evaluates the design options involved in the implementation of an event-driven circuit simulator, CINNAMON, on a general-purpose multiprocessor computer architecture. The main issues are debated, possible solutions are presented and results are antecipated. As a conclusion, the steps to be followed in the implementation, are also presented.

This work was partially supported by the Esprit 1058 *Project*

Contents

1 Hardware acceleration of circuit simulation **2**

2 Implementation issues: Evaluation of design options **6**

2.1 The Multiprocessor Architecture 6

2.2 Circuit partitioning techniques 10

2.3 Processor Management and Simulation Control 13

3 Synchronous and Asynchronous Implementations **18**

4 Implementation steps **22**

1 Hardware acceleration of circuit simulation

With the improvement of manufacturing methods and new technologies, VLSI circuits show a continuous growth in the number of devices that can be built on a single chip.

This increase in the number of devices together with the refinement in today's design technologies, makes these circuits more and more complex. This added complexity greatly explains the emerging need for circuit verification at various levels. **Circuit simulation** may well represent the difference between a circuit that will function and another that will not.

The various levels of simulation, from electrical to timing, switch-level, logic or other, provide a choice of tradeoffs between cost and accuracy. The level chosen, depends obviously on the level of verification desired: digital simulation at gate or transistor level, electrical simulation, etc, provide means for the representation and verification of the circuit functionality at different levels of knowledge. Mixed-mode simulation is another interesting approach to the problem.

Whatever level is chosen, the complexity of actual VLSI circuits requires an enormous computational effort in order for verification of large and complex circuits to be accomplished. In fact, the verification of circuits functionality, should be both reliable and fast enough to produce results in an acceptable time interval.

Several techniques for algorithm speedup were already presented and tested: these include generation of special-purpose microcode for equation solution, table-lookup model evaluation, use of vector processor machines [1], etc. **Parallelism** is however, now widely accepted as the next step to be adopted for hardware algorithm acceleration in order to satisfy the increasing demand for more computer power [2] [4] [3].

Numerous implementations of CAD algorithms on parallel processing machines, including simulators at various levels, have been reported [5] [6] [7]. Most implementations are of logic simulators [8], due to the high degree of parallelism inherent to the event-driven approach exploited in those simulators. Traditional circuit simulators do not share this characteristic and therefore do not show such a high degree of parallelism.

Behind any implementation of a CAD algorithm on a parallel processing machine, there are a number of **design options** that must necessarily be made. These design options are related to the parallelization of the algorithms, the strategy to face the problem of searching for parallelism in an originally sequential algorithm, the directions to be taken when building an algorithm whose degree of parallelism may either lead to the choice, or be forced by the architecture of the machine where the tool will ultimately lay on.

In a very broad sense, one can look at the problem from the **hardware point of view**, the **software point of view**, or both [9]. From the **software point of view**, parallel implementations of CAD tools are either the result of adaptation from sequential mono-processor versions or new algorithms exploiting parallelism on simulation techniques. From the **hardware point of view**, those implementations allow us either to define, the architecture of a special-purpose hardware accelerator, or to specify, in general terms, the global requirements for the best suited commercial architecture available.

Whichever options are made, the basic idea remains, to accelerate simulation time in order to be able to simulate larger circuits in acceptable time.

Parallel implementations of circuit simulators are no exception to this rule. Generally the hardware constraints that lead to the choice between a special-purpose machine and a parallel processing supercomputer, are dictated from the cost/performance ratio aimed at, and usually can be considered quite independent of other design options.

CINNAMON [10] [11], uses an **explicit event-driven integration algorithm** based on **voltage axis discretization** techniques. Research aiming at a parallel implementation of the algorithm on a multiprocessor machine, is under progress.

The idea is not to build a hardware accelerator, but instead to implement the simulator on a commercial machine and try to exploit the parallelism inherent to an event-driven algorithm in order to speedup the simulation.

The degree of parallelism to be achieved, depends obviously upon the **multiprocessor architecture**, since that will define the way by which data is shared and communication is performed [2]. Bus-based and non bus-based machines present different advantages and disadvantages; the choice of the architecture best-suited must be considered as an important design issue, because other options may critically depend on it.

Circuit partitioning techniques are another design option that is tightly related to the concurrency attainable in the algorithm. Other important issues are **processor scheduling, data control** and more globally, **simulation control**: **synchronous and asynchronous implementations** are suited to different computer architectures and dependent upon the partitioning strategy.

The above listed points are certainly not the only design options to be considered, but are definitely the most important ones and the ones that will dictate the improvements to be achieved with the parallel implementation of CINNAMON.

2 Implementation issues: Evaluation of design options

The mentioned design options, have a large degree of interdependence, that should not be forgotten. The relations between those options can, however, be viewed from several angles, leading to different implementations, each one stressing a different point of view and solution to a similar problem.

In the following sections, an attempt is made to analyse some possible, feasible and interesting solutions to each design option, and the restrictions that they will impose. Advantages and disadvantages of each solution will be stressed and further exploited.

The multiprocessor architecture, for obvious reasons, is bounded by several constraints, but is still a very important option. The parallelization of the algorithm [12] can be achieved at various levels leading to different degrees of parallelism. Also, global simulation control can be done synchronously or asynchronously, depending upon the architecture chosen, and influencing in different ways the performance of the implementation.

2.1 The Multiprocessor Architecture

The architecture of the multiprocessor machine for which the implementation is directed, greatly influences several design options to be taken. As the goal is not the development of a special-purpose hardware accelerator but the implementation of CINNAMON on an existing commercial machine, the number of architectures available is greatly reduced.

Multiprocessor architectures are usually divided according to the interconnection scheme used [14] [13], the mentioned constraints rule out immediately several interesting possibilities.

From structures performing **direct transfer strategy**, it is obvious that **dedicated path** configurations like **complete interconnection schemes** or **loop structures** are not in our sphere of study, because machines based upon these structures are not commercially available.

We will then restrict our study to **shared path** configurations of which **shared bus**, **shared memory**, or **shared bus and memory** structures are undoubtfully the most common and exploited available architectures.

A number of machines performing **indirect transfer strategy** are also available. Most of them are based on decentralized control methods, either **dedicated path** structures, like regular hypercubes or N-cube connections or **shared path** structures like bus-window configurations. Another approach possible would be a **centralized indirect transfer structure**, like a **star configuration**, in which the central mechanism would be the master, controlling all acitivities and downloading the data. All the attached processors would act like slaves, always dependent upon the master. This configuration is however not commercially available, because the central processor would have an enormous load compared with all the others and therefore would become a bottleneck. Other **Centralized routing** structures, like loop or bus controlled by switch schemes, have unfortunately not received much attention, due to obvious hardware limitations in processor speeds.

We can therefore divide most of the available machines in two groups: **bus/memory shared systems** and **decentralized indirect communication** structures [15] [16].

Machines of the first group present **medium grain parallelism** and are **tightly coupled systems**: examples go from pluri-processor supercomputers, to functional bus/memory shared systems, like the "Sequent Balance" multiprocessor that IMEC intends to purchase, or the "Alliant FX-8" to be installed at INESC. Sometimes called multicomputers, the machines of the second mentioned group are **loosely coupled systems** presenting **coarse grain parallelism**; examples are Intel's "iPSC" hypercube [17], the bus-window "Cm*" machine from Carnegie-Mellon University [13], or the superhypercube "Connection Machine" [16], from Thinking Machines.

The machines in the first group, that we will generically call bus-based systems, present a very desirable characteristic: as far as each processor is concerned, all other processors are alike. This type of architecture seems better suited for centralized implementations because decentralized structures face the problem of bus or memory contention. This disadvantage can however be overcome if the communication needs of all processors are minimized by a convenient circuit partitioning strategy. Bus-based systems, therefore, seem to present no inherent problem to either centralized or decentralized algorithms and hence are equally suited for synchronous or asynchronous methodologies of control.

Machines of the second group, however, do not present that characteristic. On a hypercube connection of dimension N, each processor is connected only to its N neighbours and therefore can only communicate directly with them. Message transfer to any other processor requires the knowledge of a path that leads to it, through intermediary nodes.

On bus-window systems, the situation is similar because a processor only knows directly, the processors that share its local bus. Message passing to other processor, requires the existence of external agents like bus-switches or bus-controllers. This type of architecture therefore, seems to present some disadvantages to centralized algorithms or algorithms where random message passing among processors is required.

As in bus-based structures, this problem may be overcome by an appropriate circuit partitioning technique that allows us to break a circuit into tightly coupled groups. Each of these receive inputs from just a few other groups and will output signals to a small number of other groups (at the maximum, for optimization, the dimension of the cube or the number of processes sharing the same local bus).

Although the minimization of the disadvantages found in each architectural group, is based upon careful partitioning of circuits, it is important to note that the strategies to follow, though similar, are not exactly alike. In fact, in bus-based systems, the goal is the minimization for each group of the total number of interconnections with other groups, i.e., the number of inputs and outputs of the group. On the other hand, for decentralized structures, the goal is the minimization of the number of connected groups.

In a bus-based system, for a given group of nodes, its outputs can be inputs to many other groups. If an output of a group is an input of say m other groups, the only problem that can arise, besides bus or memory contention while communicating the results of a simulation, is if one or more of those m groups is being simulated simultaneously. That problem however is not related to the circuit partitioning strategy but to the global simulation control method (the mentioned problem may even be impossible to happen in rigid synchronous implementations. This point will be debated in a following section).

In a decentralized architecture the partitioning technique must avoid problems like the one referred above. If the output of a group is the input to m other groups and $m > N$ (where N is the cube dimension), it can become extremely difficult to propagate changes on a signal because it is difficult to know where to propagate them, due to the intrinsic decentralized nature of the architecture. Even if each group were allocated to a processor for all time (a question to debate later), it is not efficient to propagate several messages through intermediary nodes along long paths.

2.2 Circuit partitioning techniques

The implementation of any CAD tool, like a circuit simulator, on a multiprocessor machine, requires the exploitation of parallelism in all phases of the algorithm.

There are two basic ways in which an algorithm can be be partitioned for an implementation in a multiprocessor environment [18].

If one allocates distinct functions as processes to each processor, either dynamically or statically, one is performing a **functional partitioning**. These functions can be allocated at the instruction level, as in a **fine-grain data-flow approach**, or in a **coarse-grain manner** allocating steps of the simulation loop to each processor. This last approach is similar to a pipelined structure where each stage of the pipe performs jobs like equation writing, model evaluation, equation solving and node scheduling activities.

Coarse-grain functional partitioning, unfortunately, is not an easily expanded solution. Adding more processors does not improve the performance, unless the number of steps is rebuilt or copies of functional units are available. Also, the interconnection network must be flexible and efficient.

The second basic way of achieving algorithmic parallelism, is by means of **data partitioning**, i.e., by having each processor perform a similar set of functions on different data items, which are allocated **dynamically** or **statically**. In this approach, parallelism can be achieved either **spatially** or **temporally**. On a circuit simulator, **spatial parallelism** requires the partitioning of the circuit in several tightly coupled groups to be simulated in parallel in the same time slot. **Temporal parallelism** is achieved by having parts of the circuit simulating simultaneously but on different time intervals.

In an event-driven algorithm, temporal parallelism is automatically achieved if we have more than one event being dealt with at the same time. Being an event-driven circuit simulator, CINNAMON presents this temporal parallelism characteristic. Also, while simulating an event on a processor, we only deal with the node scheduled and its neighbours, therefore achieving a dynamic spatial partition of the circuit.

To increase the degree of parallelism of a circuit simulator like CINNAMON, strategies for circuit partitioning must also be considered. This spatial partitioning of a network (i.e., its separation into coupled groups) is intended to minimize the expected interprocess communication, which is the inherent overhead that a parallel algorithm has over a sequential one.

Circuit partitioning can be accomplished **statically** or **dynamically**. **Static partitioning**, however simpler, is usually less efficient because it does not take properly into account the coupling level between parts of the circuit. **Dynamic partitioning**, on the other hand, is usually more time consuming but should produce better results.

Static partitioning can be implemented by a preprocessor tool. The network is separated into tightly coupled groups that will be simulated in the different processing units. Each of these groups is a set of nodes that are strongly connected, i.e., a set of nodes that is expected to suffer changes if any one of them changes value. A very consistent and valid strategy to break a network into groups was introduced by Bryant [19]. This strategy groups all nodes in bidirectional paths as tightly-connected; channel connected nodes belong to the same group. Gate nodes are at the same time outputs of a group and inputs of other groups.

This partitioning scheme, although easy to implement and approaching the real physical activity of a network as a gathering of blocks, has however a drawback: the functionality of the groups resulting from such a partitioning is not known "a priori", which is a desirable feature if we envision a network as a collection of smaller subcircuits performing a well determined function. A partitioning strategy based on a Bryant-like scheme but recognizing a certain number of sub-circuits, would be very interesting (also, mixed-mode simulation would become easier to implement in that case). For example SLOCOP [20] uses a partitioning algorithm which looks for known structures such as gates or logic units, using AI techniques.

Dynamic partitioning, can be based on a static strategy (like the one referred above), and consist in a further partition of groups of nodes into smaller classes or subsets, that are recognized to be dynamically disconnected [21]: for instance, two classes are formed from a group that has a pass-gate with a cutoff transistor. This method attempts to balance the load in the different processors in order to prevent processors from being overloaded or becoming idle.

Static partitioning is therefore easier to implement, but may fail to balance the load in the processors, which will heavily decrease the performance of the implementation. **Dynamic partitioning** may overcome this problem but care must be taken so that the dynamic division of groups into classes is done in a natural and efficient way.

In **CINNAMON**, a **fine-grain partitioning strategy** is also possible to implement and exploit. Solving an event in **CINNAMON** requires solving the scheduled node and its neighbours. This automatic partitioning of the network avoids the overhead necessary by other partitioning techniques and is fully dynamic. This strategy is inherently centralized, therefore less suited to decentralized architectures. On bus-based machines it faces bus contention problems. This innovative approach suggested by the algorithm itself, seems a very atractive technique, basically because it requires absolutely no setup-time to divide the networks into groups that are not necessary for anything else.

2.3 Processor Management and Simulation Control

The strategies for **processor scheduling** and allocation, and the scheme adopted for global simulation control, are highly dependent upon the basic architecture.

Processor allocation and scheduling can either be done randomly or follow a predefined strategy. **Random scheduling**, based on a measurement of the load in all processors, seems an attractive and simple allocation procedure. This form of scheduling seems indeed appropriate for bus-based architectures [22], due to the characteristic of this systems that all processors look alike, in the sense that communication can be established between any of them, indistinctly.

This flattened architecture, where no hierarchy is present, allows any processor to receive new data to simulate immediately after becoming idle, irrespective of its location and previous work. When a processor goes idle, it should either send a signal telling that it became free, or fetch a new group of nodes to simulate. The difference between these two approaches is related to the simulation control method implemented. For this architecture, nevertheless, measures have to be taken in order to minimize the bus contention problem.

For decentralized structures like hypercube or bus-window architectures, random scheduling presents some important drawbacks : each node in the structure only knows a few other adjacent nodes, and communication between nodes that are not neighbours (i.e., that don't share paths) is less efficient, both because the path to go from one node to the other is longer and because it requires interrupting intermediary processors to route the messages. This means that for decentralized architectures random scheduling does not seem to be a good allocation strategy.

For these architectures, other procedures have to be considered, but the same problem will arise for any dynamic scheduling scheme. On the other hand, a static scheme may overcome the communications problem, provided that the message passing needs of every processor are restrained to deliver messages only to its neighbours. This of course, is dependent upon a technique for network partitioning that would create groups of nodes in a manner that groups of nodes sharing inputs or outputs are either statically allocated to the same or to neighbour processors.

Another allocation procedure that seems well suited for both architectures is **trace allocation** [18] : in **trace allocation**, groups connected in a serial path (i.e., an output of a group is an input for another group) are placed on the same or in neighbour processors, to minimize interprocessor communication and to ease the synchronization and the data consistency checking. Other possible allocation strategies involve increasing setup times due to the overhead associated, and hence do not look as efficient.

Processor scheduling, as stated, is strongly dependent upon the multiprocessor architecture. Random scheduling is the simplest strategy for processor allocation and seems the wisest approach for bus-based architectures, but is a clumsy strategy for decentralized structures. For these last architectures, static allocation schemes, i.e., strategies in which sub-circuits of the network are allocated to a processor for all the simulation run, seem a better overall approach.

Two distinct methods for simulation control are possible: **synchronous** and **asynchronous**.

Synchronous implementations are easier to implement and debug. They usually imply the existence of a mechanism, normally a processor, that controls the simulation, schedules all other processor and manipulates the event queue.

This scheme, a master-slave relation [23], presents some drawbacks. For instance, on bus-based machines, when the master is controlling the bus, which happens every time a processor finishes its current job, all processors must wait orderly, not only for the bus to become free but also for the master to deliver him a new task. Also, if the master delivers large tasks to the slaves, which is somewhat desirable to avoid the processors being asking for new data constantly, then the master himself, may become idle for long times, therefore decreasing the performance of the system. The network partitioning is very important, because it is necessary that the groups resulting from partitioning should have sizes such that the processors finish their jobs at different times, therefore taking full advantage of bus-bandwidth.

Synchronous strategies can also be achieved through memory counters, where information about the number of processors still simulating and the sub-circuits being simulated is kept for consulting. Obviously, a mechanism to solve the problems of memory contention and to check the data consistency, must be available, but this is also true for simulation data.

This last strategy is very similar to an **asynchronous implementation** where each processor fetches a new sub-circuit to simulate but does not wait for the others to finish their simulations.

A completely **asynchronous implementation** where simulation of a group by a processor is performed as if all group inputs were correct and known for all times, can lead to a more efficient simulator, although it requires some extra data to be retained. In this quasi **chaotic** [24] [25] approach, each partition or group has associated with it a **history buffer** encoding the value of each input and of the internal nodes, for all times. Simulation of a partition proceeds based on the assumption that the current contents of the history buffer are correct. If an output of a group changes, a message is either broadcasted or directly sent to fanout groups (if one knows the processors they are allocated to). This information, once received, allows an updating of the history buffer and implies, if necessary, redoing some simulations. This strategy, requires that a **checkpoint strategy** and a **roll-back mechanism** be implemented. The checkpoint should contain all the information, node's values, pending events, messages received, etc, so that rolling the simulation back to where the checkpoint was taken, completely restores the saved state. Checkpoint strategies have therefore to be implemented; exponential checkpoint [25] seems a good choice because the probability of receiving input's variation is maximum immediately following another variation and decreases as the time since the last input's changes increases.

Asynchronous implementations seem to be more difficult to implement and debug. Also they may require extra overhead to be established, but a considerable increase in performance is expected if few roll backs are required, because simulations may proceed while slower or larger partitions are being simulated at earlier time slots.

3 Synchronous and Asynchronous Implementations

The implementation of an event-driven circuit simulator in a multiprocessor architecture requires, as stated, the definition of numerous design options that will determine its performance.

Among these options, the **architecture of the machine** is an important issue, because it places some constraints on other important subjects. Ideal **circuit partitioning techniques** [26] for once, seem to follow different criteria depending upon the architecture involved and some **simulation control methods** don't even seem appropriate for certain types of structures.

As the goal is the implementation of the simulator on an existing architecture, and not the generation of a special-purpose hardware accelerator, two basic structural architectures commercially available will be the subject of our studies: bus-based systems, that is shared-bus, shared-memory or shared bus and memory systems, and decentralized indirect routing structures, like hypercubes or bus-window architectures.

In **bus-based structures**, synchronous and asynchronous implementations seem to be both possible and efficient. Asynchronous simulation runs, are likely to better balance the load on the processing units, therefore improving the performance.

These architectures seem to be the most interesting ones for the implementation of a rigid synchronous algorithm having a master-slave structure. A completely **synchronous implementation** on a bus-based machine, could however be less efficient, if the master must wait for all the processors to finish, before starting a new set of simulations. This rigid scheme can somewhat be relaxed due to the event-driven nature of the algorithm. If two adjacent nodes in the event queue are not inputs or outputs of the group to whom the other belongs, their simulation can run in parallel without need for a total synchronization. Of course this scheme, approaching an **asynchronous implementation** could lead to some inconsistency and therefore a checkpoint strategy and a roll-back mechanism, although very small-scaled, will have to exist if the implementation is not strictly synchronous.

For these machines, **circuit partitioning techniques**, performed either **statically** or **dynamically**, also contribute to load balance, and are important requisites that should be tested. This partitioning can either be **coarse-grained** by having each node and its neighbours treated as a group for each event, a possibility offered due to the algorithmic characteristics of CINNAMON, or **fine-grained**, using an usual partitioning algorithm. Detection of well defined structures inside larger circuits is also being studied in the context of this project, and may increase the performance by decreasing the computational effort necessary for a simulation run.

For bus-based machines, **random scheduling** seems the most appropriate strategy to implement, due to its efficiency and simplicity and its therefore the option to be taken for future implementations.

The second type of machines available commercially, are machines with indirect transfer strategy and decentralized routing architecture. These machines present an obvious disadvantage for centralized-like algorithms where all the simulation is based upon an event-list, here considered as a localized controller of the flow of data. If the circuit is however partitioned in a way that load balancing is achieved, while keeping all processors to whom connected groups where allocated directly coupled this difficulty can be overcome. Note that this two constraints are in a way opposite, because it may not be possible to achieve such a partitioning with a good load balancing. This is obviously dependent on the number of neighbour processors, and if it is not possible, then the problem of propagating changes in a node's outputs may severely limit the performance attainable.

The **partitioning strategy** for an implementation on a machine with this architecture, should minimize the number of connected sub-networks in order to ease communications. The strategy for allocation of groups to processors should obviously not be random as in bus-based machines, but instead try to keep connected groups assigned to either the same or neighbour processors, i.e., should be a trace scheduling strategy. This partitioning should be static, but further partitioning the groups into classes inside the same processor is likely to become a necessary measure.

In an attempt to achieve a load balance among processors, classes resulting from the dynamic partitioning phase, can be allowed to migrate to a less heavily loaded neighbour, but the original constraint that sub-circuits are only connected to other sub-circuits on directly coupled processing units must be kept.

Due to the inherent decentralized nature of these machines, a rigid synchronous algorithm seems impossible to be efficiently implemented. Therefore possible implementations should either be completely asynchronous, implementing checkpoint and roll-back mechanisms as described for bus-based machines, or locally synchronous, i.e., each processor should wait for the neighbours that contain inputs for its groups, to tell him to start simulating the group. This strategy requires the existence of time-out mechanisms to prevent deadlock situations, and system-wide synchronization through message broadcasting to allow for data consistency preservation.

A completely different approach for the implementation of the circuit simulator, is based upon a **functional approach**. This scheme is directed to the division of the simulation loop into several steps or functions, to be linked as stages of a pipeline. This kind of implementation, inherently synchronous, although not very versatile and expandable, may prove to be efficient in certain particular structures. It can be implemented in any of the debated architectures, although for instance, not taking full profit of the possibilities of decentralized systems.

A recently suggested approach to simulation [27], models each component, a node in our case, as an **executable description**, referred to as a **model**, that is an intelligent entity in the sense that it performs the tasks of scheduling itself for execution, simulation or verification. A model schedules itself for execution when the necessary conditions are asserted at its input ports. A list of devices connected to each output is kept and changes are propagated to each of the components whenever necessary. This modeling approach, though it shows an extremely high degree of parallelism and seems extremely interesting, is beyond the scope of our present work.

4 Implementation steps

In this section, the steps leading to the implementation of CINNAMON on a **multiprocessor architecture** are briefly outlined.

Two approaches will be followed and the results obtained compared. First an implementation will be started on the Alliant FX-8 to be installed at INESC. It is a bus-based machine, with four processors in the actual configuration. Several implementations leading to different parallel versions of the simulator will be attempted.

A **coarse-grain functional partitioning** version will be developed and tested, based on a functional divison of the simulation loop. Although the best partition strategy is a matter of further research, for this kind of bus-based machines, the likely candidates seem to be: model evaluation, equation's coefficients gathering, equation solving and event scheduling activities control. In a configuration with more than four processors, then model evaluation and equation solving could be performed in more than one unit, as these steps seem to be the most expensive one's. Tests based on simulation runs of benchmark circuits will dictate the stages partitioning of this pipelined-like version.

Also, **data-partitioning** versions, with static and later with dynamic circuit partitioning will be implemented. Fine-grain and coarse-grain approaches will be followed. The node tearing or coarse-grain strategy will specially be considered because it seems the most natural way to partition a network for simulation with CINNAMON.

The versions of the simulator running on the Alliant machine, will later be carried to IMEC's Sequent Balance multiprocessor, in order to evaluate the degree of portability of the system. In a first phase, these versions will be made **synchronous**, therefore easier to implement and debug, but **asynchronous versions** will later be implemented in order to increase the performance.

A second approach will also be considered. In order to prove some ideas developed in this report, another implementation will be started on a machine with a basically different architecture.

We have chosen to run tests on hypercube structures, and Intel's own simulator of the iPSC hypercube will allow us to test a data partitioning version of the simulator on a machine with that architecture. This version will differ from the one implemented on the Alliant-like machines, i.e., bus-based, in that the static partitioning done previous to simulation start, will imply allocation of groups to processors that will remain until the end of the simulation run. Further partitioning of groups will be tested, but still on the same processing unit, i.e., no migration of groups along processors in the structure will be allowed. Test will prove if this strategy does not unbalance the structure very much, therefore reducing the performance of the simulator.

Implementations on decentralized machines seem to be more efficient if asynchronously controlled, because synchronous implementations imply a centralized master controlling the simulation. This assumption will also be somewhat verified with the hypercube implementation

References

[1] - Harro Hoepken,

"Vectorisation of Circuit Simulator ESPICE for FPS-x64",

Proc. First International Conference on Computer Technology, Systems and Applications, May 1987

[2] - Luís Miguel Silveira,

"Study of Multiprocessor Architectures for a Hardware Accelerator",

Esprit 1058 Report, January 1987

[3] - J. M. Hancock, S. DasGupta,

"Tutorial on Parallel-Processing for Design Automation Applications",

23^{rd} Proc. ACM/IEEE Design Automation Conference, 1986

[4] - Tom Blank,

"A Survey of Hardware Accelerators Used in Computer-Aided Design",

IEEE Design and Test of Computers, August 1984

[5] - Georg K. Jacob et al.,

"Direct-Method Circuit Simulation using Multiprocessor",

International Conference on CAD, 1985

[6] - Jacob White et al.,

"Accelerating Relaxation Algorithms for Circuit Simulation using Waveform Newton, Iterative step size refinement and Parallel Techniques",

International Conference on CAD, 1985

[7] - Goerge Jacob et al.,

"An empirical Analysis of the performance of a Multiprocessor-based Circuit Simulator",

Proc. 23^{nd} ACM/IEEE Design Automation Conference, 1986

[8] - Hiroshi Uno et al.,

"A Parallel Implementation of MOS Digital Circuit Simulation",

International Conference on CAD, 1985

[9] - Harold Lorin,

"Parallelism in Hardware and Software: Real and Apparent Concurrency",

McGraw-Hill, 1972

[10] - Luís Vidigal et al.,

"CINNAMON: Coupled INtegration and Nodal Analysis of MOs Networks",

Proc. 23^{rd} ACM/IEEE Design Automation Conference, 1986

[11] - Luís Vidigal, Horácio Neto,

"CINNAMON: New Results and Improvements",

to be published on the European Conference on Circuit Theory and Design, Paris, France, September, 1987

[12] - Tzu-Mu Lin, Carver A. Mead,

"A Hierachical Timing Simulation Model",

IEEE Transactions on Computer-Aided Design, vol 5, January 1986

[13] - Kai Hwang, Fayé A.Briggs,

"Computer Architecture and Parallel Processing",

McGraw-Hill, 1984

[14] - Y. Paker,

"Multi-Microprocessor Systems",

Academic Press, 1983

[15] - Paul Wallich, Glenn Zorpette,

"Minis and Mainframes.",

IEEE Spectrum, January 1986

[16] - Paul Wallich, Glenn Zorpette,

"Minis and Mainframes.",

IEEE Spectrum, January 1987

[17] - Ray Asbury et al.,

"Concurrent Computers Ideal for Inherently Parallel Problems",

Computer Design, September 1, 1985

[18] - J. T. Deutsch, A. R. Newton,

"A Multiprocessor Implementation of Relaxation-Based Electrical Circuit Simulation",

Proc. 21^{nd} ACM/IEEE Design Automation Conference, 1985

[19] - Randal E. Bryant,

"An Algorithm for MOS Logic Simulation", LAMBDA, 4^{th} Quarter, 1980

[20] - Erik Vanden Meersch et al.,
"SLOCOP: A timing verification tool for synchronous CMOS logic",
Proc. European Solid State Circuits Conference, ESSCIRC'86, Delft, September, 1986

[21] - Dundar Dumlugol,
"The Segmented Waveform Relaxation Method for Mixed-Mode Simulation of Digital MOS Circuits",
Phd Dissertation,
Katholieke Universiteit Leuven, October 1986

[22] - Project 1058 Internal Communication
"The Philips Experimental Multi-Processor System" Philips, Eindhoven

[23] - Workpackage B, Task B.b, "Esprit Proposal: Knowledge Based Design Assistant for Modular VLSI Design",
January 1986

[24] - A. Chazan, W. Miranker,
"Chaotic Relaxation",
Linear Algebra and its Applications, vol. 2, 1969

[25] - Jeffrey Arnold, Christopher J. Terman,
"A Multiprocessor Implementation of a Logic-Level Simulator",
International Conference on CAD, 1985

[26] - Prathima Agrawal,

"Circuit Partitioning for Hardware Simulation Engines",

International Conference on CAD, 1985

[27] - Sumit Ghosh,

"A Distributed Modeling Approach for Simulation and Verification of Digital Designs",

Proc. First International Conference on Computer Technology, Systems and Applications, May 1987