Parallel Implementations of Cinnamon

Preliminary Results

Luís Miguel Silveira

João Paulo Marques da Silva

INESC/IST

R. Alves Redol, 9

1000 Lisboa

PORTUGAL

Abstract

This report sumarizes the steps taken in the implementation of an event-driven circuit simulator on a multiprocessor architecture. An accelerated version of the simulator, developped in order to gain experience with the parallel computer, and the version under development are both described. This report stresses the constraints imposed by the underlying hardware, the features available, and the options taken during these tasks.

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

Contents

1 Motivation

The implementation of computationally intensive algorithms on parallel processing machines has been increasingly studied and regarded as the next step in the development of computationally intensive environments.

Many state of the art CAD tools, require large amounts of CPU power. Circuit simulators are only an example of such tools. Due to the increase in the scale of integration, complete systems are currently built on a single silicon chip, and although a "divide to conquer" strategy is usually used for verifying them, a global view is very important and requires the use of time consuming CAD tools. The costs of production, including design, the probability that the circuit will work and the turnaround time from project to foundry are greatly enhanced if proper simulation tools are available.

CINNAMON [1], a circuit simulator being developed at **INESC**, is based on new techniques for simulation control, in order to achieve a high performance. The aim of the simulator is to take advantage of the latency pervasive in circuits to dramatically increase the speed of simulations.

The steps taken and the techniques used in order to achieve that speedup, have been described elsewhere [1] [2], but will be summarized later. It is important to note however, that the efficiency of the techniques used in the simulator vary with the size and the activity of the circuit being simulated. For small circuits the results obtained do not, as expected, show large improvements. Still, even for very small circuits, **CINNAMON** keeps up with traditional simulators in terms of speed and accuracy, while for larger circuits, large speedups have been obtained, depending only on the activity of the circuit.

One of the main ideas developed in **CINNAMON** is the concept of **localized activity**. It is fairly obvious that changes in one area of the circuit take some time to propagate to other areas. This concept has lead to the use of event-driven techniques in order to control the process of simulation.

Based on this concept of **localized activity**, it is evident that if two or more fairly apart areas of a circuit, suffer "changes" at a certain time, those changes could be evaluated in the neighborhood were they occur and only later propagated to nearby areas. This thought suggests that these areas, where primary "changes" occur "simultaneously", could if processors were available, be simulated in parallel and the results later propagated to the rest of the circuit, hence somu gain is to be achieved by implementing the algorithm in parallel architectures.

This report describes the efforts made in that direction. It consists of six sections. In the next section the architecture of the parallel computer, an Alliant FX/4, is described and a feeling to its possibilities is given. Section 3 briefly describes the algorithm of the circuit simulator **CINNAMON** developped at **INESC** under Project Esprit 1058, and used as core for this task.

Section 4 summarizes the first attempt of an implementation of the algorithm on a parallel processing machine. This version should only be viewed as an accelerated version of the simulator, which allowed us, nevertheless, to get a good understanding and some experience with the Alliant machine and its features.

Finally section 6 describes the prototype of a data-partitioning parallel version of bf CINNAMON. Particular stress is given to the setting of the data-strucures and the simulation control algorithm. The report concludes with a few conclusions of our experience so far in this task.

2 The Hardware: Alliant FX/4

The Alliant machine installed at INESC is an FX/4 mini-supercomputer. In architectural terms, it is an **MIMD** machine, shared-memory and bus-based, with vector capabilities [3] [4]. With a total of 6 processors the Alliant is a high performance vector-parallel machine [6].

With 4 computational elements (expandable to 8) or **CE**'s, linked through a high-speed crossbar interconnection network to cache memories that ensure faster access to data and code while keeping memory coherency to all processors, the Alliant FX/4 achieves up to 5 MFLOPS peak performance on Linpack benchmarks (manufacturer data).

Two additional processors (expandable to twelve), the interactive processors or **IP**'s, linked through caches to main memory, deal with interactive computing thus maintaining high availability and network response, while enabling the **CE**'s to concentrate on computation intensive jobs. These processors basically execute interactive user jobs, operating system tasks created by jobs running on the **CE**'s, like I/O, and general system activity like paging.

An arbitrary number of **CE**'s can be defined at boot time to form the computational complex, where parallel jobs will run. In this mode of functionment all **CE**'s defined to be part of the complex can only run parallel tasks. However, another mode exists, where the complex is formed, kept for a certain amount of time running parallel tasks, and then blown, and all **CE**'s become detached for a certain time interval. The procedure is then periodically repeated. The scheduling of time for complex formation and the time interval when **CE**'s will run in parallel can be defined and changed with no need for system reboot. Also, in both modes not all **CE**'s need to be part of the complex: some can be kept detached, the decision being left to the system supervisor according to local necessities. The machine on which the studies being described were developed, was configured in a dynamic mode, three **CE**'s complex.

Although its reported performance can be considered exceptional, the machine has mainly been developed for computationally intensive scientific calculations, rather than as a general parallel multi-user multi-computer system. The difference can be seen by the number of possibilities available for the programmer to fully profit from the existing parallelism.

A high performance "FORTRAN" compiler allows use of vector and concurrent facilities to boost otherwise time-consuming code. These facilities are however not directly available to the programmer and they are still very much fine-grained (at the instruction level, like optimization of "FORTRAN" DO loops, etc). A similar "C" compiler has been announced and will be available soon.

A "C" programmer has access to a "concurrent call" feature that allows subroutines to be executed in parallel. A pool of vector and concurrent functions is also available for scientific calculation [7]. General operating system mechanisms supporting shared memory, usual forms of synchronization and atomic locks are also available.

The work described in this report consisting of the parallel implementations of **CINNAMON** was done in "C", since the original code of the simulator was also written in "C" which seems a very appropriate language.

A single operating system feature allows the programmer to take advantage of concurrency. It's the **concurrent_call** which has the form:

$$\textit{concurrent\_call(call\_flag} \mid \textit{quit\_flag, func, n, } arg_1, \ldots, arg_n)$$

A **concurrent call** issued from a process executing a "C" program creates "n" tasks in the context of the process and schedules each task to an available processor. The "func" function is typically

/ function called by concurrent_call */*

func(nc, $arg_1, \ldots, arg_n$*)*

int nc ; / # of call */*

...

{

code for func ;

}

Note that "func" receives as its first dummy parameter the number of the call that it is processing (calls are named "0" through "$n-1$").

There are three basic ways of calling **concurrent_call**:

- if *"call_flag"* = *"CNCALL_COUNT"* a constant defined in *"cncall.h"* : the system will create "n" tasks to run on the computational complex, which are scheduled to start when processors become available

- if *"call_flag"* = *"CNCALL_NUMPROC"* : the system will create one task for each existing processor in the complex. In this case, the count argument ("n") to **concurrent_call** should be omitted

- if *"call_flag"* = *"CNCALL_INFINITE"* the system will continuously create tasks each calling "func". It is up to the programmer to provide mechanism to explicitly quit this infinite loop and make sure that the program will eventually end

A **concurrent_call** can terminate explicitely or implicitly by appropriately setting the *"quit_flag"*:

- if *"quit_flag"* = *"CNCALL_NO_QUIT"* the concurrent call will terminate only after all tasks were created and finish

- if *"quit_flag"* = *"CNCALL_NONZERO_QUIT"* or if *"quit_flag"* = *"CNCALL_NEGATIVE_QUIT"* the concurrent call will return when one of the tasks returns a nonzero (or negative) value

Primitives exist (*quit_enable()* and *quit_disable()*) that prevent or allow the complex to be blown at a certain time. These primitives should surround code that the programmer knows should not be interrupted. In normal functioning, however, the **concurrent_call** waits for all tasks (or only those already created) to terminate, in order to proceed, therefore providing with a kind of synchronization-on-finish primitive.

Two other primitives, from a "common library", are of interest in the context of this work:

lib_number_of_processors()

that returns the number of **CE**'s defined to be part of the computational complex (thus allowing programs to become independent from a particular configuration), and

lib_processor_number()

that returns the number of the processor where the task is running. Both functions are to be used only in concurrent mode, i.e., they should only be called from within concurrent calls (otherwise they return 0).

Usual facilities for shared-memory are also available. However, for anyone using task parallelism (i.e. calling **concurrent_call**), these features aren't of much interest since all tasks created trough the concurrent call primitive co-exist in the context of the calling process and directly inherit access to all its memory space. The possibility of defining shared-memory regions can nevertheless be of interest for certain applications.

Synchronization mechanisms are essential in such a machine. Atomic locks (and unlocks) can be used either for a pool of tasks or processes sharing access to some data. These standard library routines are indeed the basis for task synchronization in the machine since any other mechanism (message passing, signals, etc) seems awkward because the tasks co-exist in the space of a single process and all the process memory is shared by all tasks.

Some performance tests were carried in the Alliant, to evaluate the performance of task creation, data locking and unlocking, etc: The fact that task generation is extremely fast (about 400 times faster than creating a son process with the system call "*fork()*")

is a good indicator of the level of performances that can be achieved on the machine. In the Alliant, task generation takes about 2 $\mu secs$; the operation of locking and unlocking a memory position is also extremely fast (about 40 $\mu secs$).

The work described in this report is built around concurrent call's, using locks and unlocks as task synchronization primitives.

3 The Circuit Simulator

CINNAMON is an event-driven electrical circuit simulator based on voltage axis discretization techniques. An event is caused by a change in a circuit node. To evaluate an event means to "predict" the future consequences of the node's change in the state of the node itself and its neighbors.

All node events are kept in a list. Each node appears in the list just once (except for voltage sources that are scheduled for all times at the beginning of the simulation, and in the latest versions, use a separate list).

The information in each event structure says that node "k" will be at voltage "v" at time "t". When time "t" is reached, the node voltage is updated to "v" and the effects of that change are propagated. Consider that at time "t_o" node "k" changed to "v_o". The nodes matrix equation of the circuit.

$$[C]\,[\dot{v}] + [G]\,[v] = [\imath] \tag{1}$$

leads to an equation for node k in the form of

$$c_{kk}\dot{v_k} + g_{kk}v_k - \sum_{j\neq k} c_{kj}\dot{v_j} - \sum_{j\neq k} g_{kj}v_j = \imath_k \tag{2}$$

(where the sums are for all nodes in the circuit, except k, c_{kj} refers to a capacitor connected between nodes k and j, g_{kj} is similar for conductances and $\imath_k$ is the sum of all current sources entering node k).

The current approach in **CINNAMON**, decouples all nodes. The $n * n$ system of equations of the circuit leads to:

$$c_{kk}\dot{v_k} + g_{kk}v_k = \imath_k + \sum_{j \neq k} c_{kj}\dot{v_j} + \sum_{j \neq k} g_{kj}v_j \tag{3}$$

The simplest approximation is to use the last estimate for $\dot{v_j}$ and v_j. A good approximation to the solution of the linear differential, considering that the node voltages do not vary significantly in the time interval under consideration, is then:

$$v_k(t) = (v_{k\infty} - v_{ko})e^{(t-t_0)/\tau_k} + v_{k\infty} \tag{4}$$

where $v_{k\infty} = i_k/c_k$ and $\tau_k = c_k/g_k$. Whenever this approximation is considered valid, we can easily find the time when the node will reach the next voltage level $(v_{ko} + \Delta V)$ at time

$$t_{next} = t_0 + \tau_k \ln(\frac{v_{ko} - v_{k\infty} \pm \Delta V}{v_{ko} - v_{k\infty}}) \tag{5}$$

and schedule the new event for the node. The change is then propagated to the adjacent nodes by scheduling new events for them.

The simulation is controlled by the list of events, and is considered to be terminated when no more scheduled events are found in the list. Issues like equations for device modeling and parameter calculations do not differ from other versions, and will not be mentioned here, but they can be found elsewhere [8].

4 An Accelerated version of Cinnamon

This section describes the steps taken and results obtained in the development of an accelerated version of **CINNAMON**.

This version was the first attempt of an implementation of the algorithm on a parallel processing machine. This version is not based on a parallel algorithm for **CINNAMON**. In fact it should only be viewed as an accelerated version of the simulator. The difference will, hopefully, be understood in the next sections.

From a certain point of view, this version can be understood as a data-partitioning implementation [5]. However, as will soon be explained, the partitioning of data in this version is only performed at very well defined and precise places of the simulation loop.

4.1 Simulation Control Algorithm

The simulation control algorithm of the sequential uniprocessor standard **CINNAMON** is simply:

```
simulation() /* either DC or TRAN analysis */
{
   time = 0 ;
   schedule voltage sources ;
   while(there are more events and time < Maxtime)  {
       get next event information (ev_node, ev_time, ev_voltage) ;
       time = ev_time ;
       linearize devices connected to node (ev_node) ;
       evaluate next change (ev_node, list of adjacents) ;
       print information(ev_node, ev_time, ev_voltage) ;
   }
}
```

In this version, the essential sequential algorithm was kept, but parallelism was seek at a very fine-grain, by looking at actions that could be performed in parallel.

By looking at the basic simulation loop just presented, we could immediately detect the existence of three different types of actions related to the "solution" of an event.

An essentially sequential part that deals with getting the information related with the event to be processed and later on with the recording of its results. A section related to the set-up necessary, prior to the calculation of the elements that appear in the circuits system of equations. A third section related to the calculations that are necessary when solving an event, and which may cause new events to be scheduled.

These three sections of the simulation loop were dealt with in different ways. The sequential part of it, was left untouched since little was to be gained by any change in its structure. Nevertheless, it is worth saying that the printing of the results can in some cases, represent a considerable fraction of the time spent in the simulation.

The setting up of the elements on the circuit equations deserves a bit more of attention. Since the current algorithm in **CINNAMON** decouples the circuit's equations at the time of each event, only the elements appearing in one of the equations (the one for the node that caused the event) are needed. Even in this simplified case, this requires the linearization of all the circuit elements that are connected to the scheduled (or active) node. Note that this is essentially a sequential task for each device, which can, in principle, be linearized separately (therefore in parallel) from any other one. We see therefore, that parallelism at this level can be of help and brings very little additional effort, since in this case there is no data-dependency to check upon.

An event in **CINNAMON** is always related to changes in a node, thereafter called active. The changes in the node's voltage correspond to perturbations that may lead to further changes in the node itself and its neighbor nodes. This perturbations, therefore, have associated to them a concept of propagation. This means that changes in a node, may propagate to its vicinity and henceforth to the rest of the circuit. Essentially, this implies that when solving an event for a node, the node and its neighbors have to be calculated, in order to ascertain if the perturbation is propagating or not. Once again in the current algorithm, this can be performed in parallel, therefore leading to increased reduction in simulation time.

There is however a significant difference in this last case. The fact that a perturbation may propagate to the vicinity of an active node, means that when solving an event, new events on the node itself and its neighbours, may have to be scheduled. As the event list is unique for each simulation run, this implies that different processors, may try to change the information on global data, by introducing (or extracting) events from the event list.

To ensure data consistency, a method is needed that provides for synchronization in the access of the event list. In the present version mutual exclusion in the access to global data is ensured by the use of atomic locks. The use of this lock variable can be understood as if the processor were trying to lock the list itself so as to get exclusive access to it. When a processor wants to change the structure of the event list it must lock the list in order to be able to proceed. If when trying to lock it, it discovers that it is already locked, it just sits back and waits for the list to be unlocked.

The algorithm for the present version as it stands, is therefore:

```
simulation()
{
   time = 0 ;
   schedule voltage sources ;
   while(there are more events and time < Maxtime)  {
       get next event information (ev_node, ev_time, ev_voltage) ;
       time = ev_time ;
       node = ev_node ;
       devices = number of devices connected to the scheduled node ;
       concurrent_call(..., LinearizeElements, devices, node) ;
       neighbours = number of neighbours of scheduled node ;
       concurrent_call(..., EvaluateNode, neighbours+1, nd) ;
                                     /* +1 for the node itself */
       print information(ev_node, ev_time, ev_voltage) ;
   }
}



LinearizeElements(proc,nd)     /* linearization of elements prior to
                                  solving the equations */
int proc, nd ;
{
linearize the device number proc in the list for node nd ;
}


char EventList ;                /* global semaphore variable for
                                   synchronization in the access to
                                   the event list */



EvaluateNode(nn,nd)            /* evaluation of perturbation
                                   propagation  */
int nn, nd ;
{
n =  node number of nn-th neighbour of nd ;
evaluate next change for node n ;
t = time of next change for n ;
lock(&EventList) ;
schedule node n at time t ;
unlock(&EventList) ;
}
```

Another acceleration technique that was also used, was to detect the existence of computationally intensive portions of code that presented no data-dependency. These could be parallelized easily. This approach, was carried out through a careful look at "profiles" from the execution of the simulator and eventually lead to a decrease in simulation time by having some time-consuming functions to be executed in parallel. Setting up the data structures for simulation was one of the parts of the code were this technique was used.

The next section will address some of the results obtained thus far with the version just described.

4.2 Results for the Accelerated version

The basic trends for this accelerated version of **CINNAMON** were described in the previous section.

The results obtained so far can be considered as acceptable. The acceleration factor porsued has in fact been obtained. In the following table, the results obtained for this version, for a series of benchmark circuits are presented. These measurements refer to the transient analysis on an Alliant FX/3 (3 processors in the complex).

The circuits tested are very dissimilar in the hope of getting a better view of the possibilities of this version. A short description of them follows:

- "100inv": a chain of 100 CMOS inverters
- "2gi": a series of two inverters and pass-gates with Grenoble models

- “2nc”: two NMOS inverters
- “a4”, “alu4”: 4-bit Mead & Conway ALU
- “a16”, “alu16”: 16-bit Mead & Conway ALU
- “a32”, “alu32”: 32-bit Mead & Conway ALU
- “bi02”: bootstrapped inverter
- “compt”: 4-bit comparator
- “fault”: a faulty chain of inverters
- “inv2”: two CMOS inverters
- “inv4”: four CMOS inverters
- “mem4”: a four bit memory element
- “nand5”: a five input nand gate
- “ring1”: a 19-stage ring oscillator
- “senser”: a sense-amplifier

Transient Analysis Simulation Time (secs.)

Circuit	v4(VaxStation)	v4(Alliant)	v4(Accelerated)	speedup
100inv	1:27.96	20.02	9.78	2.04
2gi	15.40	4.15	3.20	1.30
2nc	51.27	13.99	11.38	1.23
a16	8:57.68	2:27.78	1:08.42	2.16
a32	8:45.50	2:21.80	1:03.98	2.22
a4	1:05.32	17.82	8.24	2.16
alu16	4:49.21	1:20.04	42.23	1.90
alu32	36:15.44	10:47.92	5:20.65	2.02
alu4	4:25.51	1:17.15	38.87	1.80
bi02	9:09.92	2:50.08	2:16.81	1.02
compt	16:15.29	4:23.45	2:15.80	1.94
fault	3.18	0.83	0.71	1.17
inv2	5.39	1.54	1.34	1.15
inv4	10.01	2.72	2.05	1.33
mem4	37.61	10.51	6.55	1.60
nand5	12.53	3.41	2.31	1.48
ring1	56.17	14.35	8.23	1.74
senser	18.47	4.58	3.06	1.50

The results presented in the previous tables deserve a more detailed inspection. Roughly speaking, a factor of two has been obtained for most medium-size circuits, like the alu's and the comparator.

One should not forget that in this accelerated version, parallelism has only been achieved at a very fine-grain, at the loop level. Therefore, even with three processors (the configuration of the Alliant machine used), a 50 % speedup can already be considered a good figure, considering that accesses to the event-list are sequential, and that at least half of the processing needed to solve an event is sequential: extract the event, determine the time and node that caused the event, plotting the event's data and accessing the event-list again for new scheduled nodes (which can happen several times for each event, depending on the connectivity of the node causing the event). Plotting the data for an event may take as much as 25 % of a normal simulation time, for certain circuits. The sum of all these sequential actions is the real bottleneck for a greater efficiency achievement.

It should be noted that in the version just described, parallelism was seek at the "linearization" and "event-solving" steps. In the first case, the potential speedup achievable is bounded by the average number of non-linear devices connected to a node. Also, in the second case, the possible speedup is bounded by the average node connectivity. In both cases, therefore, it should be clear that the speedup obtained is not scalable with the number of processors, but is dependent (and bounded) by the characteristics of the circuit being simulated. This version, therefore, is better suited for machines with a small (up to four or five) number of processors.

It is our belief that the goals intended with this version were fully obtained. In fact the development of this version gave us a greater insight into the architectural features of the Alliant computer, the way to explore its potential and to obtain an even faster simulator without greatly modifying the original algorithm of the simulator. With this in mind we believe the work has been rewarding in many ways.

5 A Data-Partitioning version of Cinnanmon

The version of **CINNAMON** described in this section is based on Data-Partitioning among the available processing units.

Our approach partitions the data into "tightly" connected "sub-networks". Each processor will simulate a "sub-network" while a set of conditions is met. When these conditions are no longer met, the processor stops the simulation of the sub-circuit, drops it and grabs the next sub-network able to be simulated. This is the basic procedure for simulation control. Obviously, access to all shared data must be synchronized to ensure data consistency.

The next section deals with the partitioning strategy. We will at first use the term "sub-network" instead of the more obvious "group". The reasons for that will soon be understood. The two following sections explain in more detail the simulation control algorithm and the concepts introduced with this version. This version is still in a preliminary form and further test and debugging are to be performed next.

5.1 The Partitioning Strategy

CINNAMON's Parallel version 1 is based on a **static data partitioning strategy**. The basic ideia for this partitioning is closely related to the event-driven nature of the algorithm and tries to explore a little further the concept of **localized activity** [5]. Changes in one area of the circuit cause, at first, only local disturbance, that will take some time to propagate to further areas. This concept leads to a natural way of solving the equations and to the use of event-driven techniques to control the process of simulation [2]. If multiple localized activity centers exist, they can be dealt with in parallel with little or no interaction.

In a previous report, [9] a static partitioning tool was described. This version uses that tool to partition the circuit into tightly coupled "sub-networks". The main ideias porsued in the partitioning scheme are now described.

On one hand it is easily seen that if the result of data partitioning is a set of strongly connected "sub-networks" with a minimum of interaction, then internal changes in one "sub-network" can be dealt with locally and only later will appear on the "border" of the "sub-network". Only then, they will have to be propagated to the "neighbor sub-networks". This is the main ideia on which data-partitioning techniques are based.

The partitioning strategy implemented in the pre-processor tool was based on the "disturbance locality" concept introduced. Communication between sub-networks is therefore only needed when that disturbance has to be propagated to neighbor sub-networks.

An essential concept for this partitioning strategy was introduced, the **network border** which is the set of nodes from a sub-network that have connections to foreign nodes, i.e., nodes in other sub-networks. A node belonging to a **sub-network border** is therefore called a **border** node. Nodes that are connected to **border nodes** and are not themselves in the **network border**, are called **nearborder nodes**. Voltage sources have a special tag, since their changes are known beforehand.

After the partitioning, the circuit is divided into four kinds of nodes: **external**, **internal**, **border** and **nearborder**.

The higher level of the data hierarchy is therefore the sub-network. We need, for each **border** node a list of connected sub-networks, which will allow changes in a sub-network, that arise in its border, to be propagated to the rest of the circuit. An array of structures is created at partitioning time containing, for each node, the number of the sub-network it belongs to, the type of node and, if it is a **border** node, a list of neighbor sub-networks to be warned when that node changes.

5.2 General Concepts

The simulation control algorithm used in **CINNAMON** is:

```
simulation()
{
   time = 0 ;
   schedule voltage sources ;
   while(there are more events and time < Maxtime)  {
       get next event information (ev_node, ev_time, ev_voltage) ;
       time = ev_time ;
       linearize devices connected to node (ev_node) ;
       evaluate next change change (ev_node, list of adjacents) ;
       print information(ev_node, ev_time, ev_voltage) ;
   }
}
```

In our implementation we require a more tricky control of simulation. We have several processors competing to grab the data, and mechanisms have to be provided to allow only one processor at a time to change each set of data. Also, with the event-driven algorithm, all processors will be simulating at different times and data consistency must be enforced so that slower processors do not get corrupted data from faster processors.

Circuit simulators based on time axis discretization do not share this last problem. Indeed, even if data is partitioned and each processor calculates a set of data, they are all working in a lock-step fashion and, at worst, they will have to iterate at a certain interval so that data is consistent and they can proceed to the next time interval (for instance Waveform Relaxation techniques can calculate waveforms for each set of data separately and then iterate on the results).

In spite of this problem, many algorithms using event-driven techniques have been implemented on multiprocessors. However, most examples reported, are of logic simulators, which present two extremely important characteristics: on one hand they imply unidirectionality, and, on the other hand, parallelism is exploited at each time-point, which implies a lock-step action among all processors.

When considering the choices for simulation control, one can easily see that the usual techniques used for instance in logic simulators do not apply here. In most parallel implementations of event-driven algorithms, a precedence constraint is enforced in that a group or partition or set of data, can be evaluated at time $t + \Delta t$ if all its inputs are know at time t. The counterpart for a simulator like **CINNAMON** is not obvious since there is no notion of inputs to a partition (except for external inputs that present no problems) and it is not possible to decide when to solve an event based on the state of all adjacent nodes since at least some of them will have their last change in the past. In such a scheme, at best, one single node could move, i.e., a sequential version, because only the slowest processor could solve its event.

The solution to this problem is to consider it from a different point of view. If the value of all nodes was known for all time, there would be no precedence constraints to enforce and each partition could be simulated independently of all others [11]. In this sort of **chaotic simulation**, each partition proceeds with its simulation based on the assumption that all values it will use are known for all time. Whenever a change occurs, all partitions that may have to use the new value will have to be warned and may have to redo their calculations if they find out that they did it with wrong data.

In order to do this, they must have a record of the data used and a **checkpoint strategy** must be defined so that a partition can store its state at certain time intervals. A **rollback procedure**, will allow a partition to go back in time, to recover, from the list of checkpoints, the correct status before the wrong calculation was done, and to proceed.

In our **data-partitioning** implementation this strategy is followed. For each of our groups, a checkpoint list is kept with the record of all changes in the state of the group's nodes, and mechanisms for rollback were developed.

An important difference should be noted however. In the groups resulting from the partition, all border nodes are at the same time inputs and outputs. If a border node of a group is changed, then all its neighbors, connected to that node, are warned and maybe will have to rollback their status. Each neighbor, after rolling back its simulation, will redo the calculations on its side of the border and if a change occurs it will also warn its neighbors including the group that first warned it. This sequence can be rather nasty, particularly if the first group has continued its work, or if border nodes on both sides suffer great changes and the voltage step is small. It is also implicit that there will be multiple copies of all relevant information concerning border nodes, one for each side of the border, to ensure consistency.

In order to avoid this multiplicity of data, and also to avoid the multiple rollbacks when a border node changes, we introduce a constraint the we will call the **"Border Constraint"**: *when, inside a partition, an event occurs in a border node, the event will be solved if and only if the partition is the holder of the global time, i.e., if it is the slowest of all partitions.* Under this constraint when an event on a border node is to be solved, the processor handling that partition has to make sure that all other partitions (the ones scheduled and the ones being simulated) and the next voltage source event, are all ahead of it in time. If that is the case, it will solve the event; otherwise, it is better to drop that group and grab another one (a much slower one may be waiting to be simulated).

The strategy described has important and useful consequences besides the ones already referred. By having only the slowest group solving border events, then a group that discovers it is the slowest one at time t, immediately knows that all its "past" is correct and that all calculations until t are also correct for all other groups (if he is the slowest, no one can go back to a time previous than t, so they are all correct until t).

This is a disguised, though efficient form of synchronization (auto or local synchronization) between groups. It is automatically performed and does not require as usually, that all processors stop for global synchronization to be done.

Each group proceeds internally without even looking at other groups. When a border event is solved, its neighbors must be warned of that change. Note that when solving a border event one uses and changes data that is not strictly ours (it is shared), so only one processor can do that at a time.

This implies another concept: that of warning neighbor groups that they should see if they have not done wrong calculations. Instead of warning, we rather chose the word **Order** to qualify this message passing mechanism. The reason for this choice is that the originator of the message is indeed ordering the receiver to do a calculation and he will not have to go back, but the receiver may have to. Note that if a group discovers at time t that it is the slowest, it sends an order consisting of the time, t and the node that changed. The fact that the order keeps the time (which is quite obvious that it should) implies that each node only needs to have space for one order, (provided that when determining the time of a group one looks to whether it has any pending order). This is a consequence that after receiving an order at time t no other group can send it another order at a time greater than t, because that would be a violation of the **Border Constraint**.

An explanation is due about what an order means: when we send an order after solving a border event we have just rescheduled the border node and some of its adjacents, but we do not reschedule nodes that do not belong to our group, instead we tell its owner that he should perform that calculation. The only difference between solving an order and solving an event, is that, when solving an order, only the node itself is rescheduled, not its adjacents.

By receiving an order, a part immediately synchronizes with the slowest processor and knows that all its calculations are correct until the time of the order. This can be used to output the results.

Now that the basic concepts have been introduced, we describe some design options and the control algorithm.

5.3 Control Algorithm

The simulation control algorithm for this data-partitioning version is still based on event-driven techniques. However, rather than a global list of scheduled nodes, we now have an list of groups, sorted by increasing time (minimum of time of order and time of first event).

Whenever a processor becomes free, it grabs the first group in the list and advances the list. The strategy for processor scheduling to groups is random, which is the fastest one and in this case does not present any disadvantage. When a processor drops a group it inserts it in the list, in the proper place, if the group still has events or has just received an order, or otherwise simply drops it, leaving it unscheduled. Access to the list of groups is done through a "gate" or "semaphore", consisting of an atomic lock that ensures that global data is not changed by two processors at the same time and it is not changed when someone else may be reading it.

Besides having a list of neighbor groups attached to every border node, it is also necessary to have some volatile information attached to each group: the number of the group, a flag stating the status of the group (simulating, scheduled or unscheduled), the number of the processor that is simulating it (when being simulated), a list of events for the nodes belonging to the group, an order structure functioning as a mailbox address where neighbor groups can leave orders, and a "semaphore" to access it, a list of checkpoints from where to recover past status if necessary and a "semaphore" to access it.

These internal "semaphore" structures are the means by which access to global data is synchronized. They are changed through the "lock" related system calls.

There is also another structure that must be kept besides the list of groups and the list of events for each group. It is related to the state of the processors and should give the status of the processor (running or idle), and contain the information (number) of the group it is simulating, when it is not idle. This information is useful when determining the slowest group and also to determine when to return from concurrency once the simulation is finished.

In order to do the implementation being described, basically 80 % of the simulation kernel had to be rewritten (i am not taking into account the code related to the input parser and the model calculation routines, which are not directly connected with the control of simulation but nevertheless are more than 50 % of the code). The routines handling the groups and the group list were written from scratch; those dealing with node events merely changed. The basic routines of the simulation control algorithm are completely new, from high-level processor scheduling to low-level event and order solving.

An overall description of the algorithm implemented follows: routines that are not described but merely mentioned have obvious functionality.

```
/* CINNAMON Parallel version 1
 *
 * Data Partitioning Coarse-Grain Parallelism
 *
 * Author: Luis Miguel Silveira (INESC)     1987-1988
 *
 */

/* main()
 *
 * main simulation loop. determines high level scheduling of work
 * and creates the concurrent tasks
 *
 */

int NumProcs ; /* the number of processors in the complex */
int GroupNumber ; /* the number of groups */

main()
{
    determine and open input file ;
    parser of input data ;
    build internal structures ;

    /* sets NumProcs to the number of the processors in the complex */
    concurrent_call(..., getNumProcs, 1) ;  /* 1 call is enough */
    partition the circuit;

    do_DC_analysis();

    /* start TRANSIENT analysis; set environment */
    set TRANSIENT analysis ;
    open TRANSIENT results file ;
    Schedule all time varying sources ;


    /* to really start analysis 'solve' first source event */
    SchedInputOrder() ;  /* will order i.e. schedule neighbor groups */

    concurrent_call(..., simul, NumProcs) ;
    close TRANSIENT file ;

    simulation statistics;
    end simulation ;
}
```

```
/* simul()
 *
 * simulation loop. runs in parallel mode in the complex. determines
 * scheduling of processors to groups.
 * calls the event solver routines. receives a dummy parameter
 * telling the number of the processor it is running in
 *
 */

simul(np)
int np ;
{
    int Anymore ;

    for(;;)  {
        for(;;)  {
           if( MoreGroupsScheduled() )  {
               get a group and keep its number ;
               /* the group status is now 'simulating' */
               set processor status to running ;
               AnyMore = ProcessEvents(group) ;
               if( AnyMore ) insert the group in the list ;
               else simply drop the group ;
           }
           else break ;
        }
        set processor status to idle ;
        if( all processors are idle )  {
            if( MoreGroupsScheduled() ) continue ;
                                    /* there is more work */
    else if( there are source events scheduled )  {
                 SchedInputOrder() ;
                 continue ;
            }
            else return ;  /* nothing more to do ;
        }
    }
}


/* ProcessEvents()
 *
 * group internal loop simulation. determines whether orders or
```

```
 * events are solved and calls the necessary routines
 *
 */

ProcessEvents(group)
int group ;
{
    int status, node_order ;
    double time_order ;

    do  {
       if( there is a pending order )  {
           get data for the order ;
           SolveOrder(group, node_order, time_order) ;
       }
       if( there are no more events scheduled ) return(no more) ;
       status = SolveEvent(group) ;
    }
    while( status is can do ) ;
    if( status is can't do ) return(there are more) ;
    else return(no more) ;
}
```

The simulation control algorithm has just been presented. The next section describes the strategy implemented for checkpoint, how to use that information to recover the status of a node or a group at a certain time, and how and when to Rollback and until what time.

5.4 Checkpoint and Rollback Strategy

The checkpoint list is a list of all relevant information that can allow simulation to rollback and status of nodes to be recovered. It should therefore have all the necessary information related to actions taken during the process of simulation.

In our implementation, besides the event list, there is also the notion of "order", as a means for simulation control. However if the **"Border Constraint"** is enforced, then there is no need to keep information about orders. Rather orders allow us to clean the checkpoint list because as we saw, they act as a synchronization mechanism.

In our algorithm we decided to store only information about the changes that were made in the event list of the group, rather than filling it with a periodic "photography" of the group.

The only information that we have to keep then are the changes in the event list, not only for events solved, but also for those that where scheduled or unscheduled (i.e., removed or inserted in the list). The checkpoint does not therefore need to be periodic (as in other algorithms [11]) but simply has to be made whenever we insert or remove a node from the event list (note that removing a node from the list for solving an event is quite different from removing a node but not solving an event, because in the first case a node's value may be changed).

With this scheme, the checkpoint strategy becomes therefore quite straightforward and very obvious to use the checkpoint data to recover past status.

If we want to recover the status of a node at a certain time, we proceed as follows: we look at the global data attached to the node and see when was its last change; if it was before the current time, then the value is correct and we need do nothing more. If however the node changed after t, we must go through the checkpoint list of the group the node belongs to, starting from the end, tracing back all changes for that node after t.

The decision if and when to rollback, is a bit more tricky. First we note that until an order is received, a group ignores all other groups, and acts as if it were the only existing one. Rollback is therefore only needed after an order is received. The actual procedure is as follows: when there is a pending order for the group, a “BackStatus(time)” function is called which goes through the checkpoint list from the end until the current time and recovers the status of all group nodes at the time indicatd in the order.

Therefore, when the order is solved, the status of the group is already correct at “time”. But solving an order will imply scheduling or unscheduling at least one node. The (un)scheduling routine (for nodes) will, before operating on the event list, and if an order is being solved, will perform a “Rollback”.

To rollback is simply to recover the event-list at the proper time (the minimum of the time of the first event for the node (un)scheduled and the time of (un)scheduling) and to get ride of the checkpoint list thereafter.

The last operation when solving an order is to forward the status until the time the rollback was performed (a procedure dual to “BackStatus()” and obviously called “ForwardStatus()”).

“BackStatus()”, “ForwardStatus()” and “Rollback()”, are the only procedures that deal with the checkpoint list.

6 Preliminary Conclusions

An accelerated version of the circuit simulator **CINNAMON** has been described, together with a data-partitioning implementation on a parallel machine. The basic algorithm for this data-partitioning coarse-grain parallel implementation of CINNAMON has been described.

The last version currently being developed is still in a preliminary status for conclusions to be drawn.

This algorithm doesn't take much advantage of the capacities of the machine. Its high speed in creating jobs isn't much of a help since in the implementation described we create at the beginning of simulation as many tasks as processors and they only finish when simulation finishes. The ability to create tasks in the context of one process is however of importance, since that allows us to avoid process communications that will no doubt be more time consuming (the only possible way to achieve similar performance with processes instead of tasks would have to rely on the use of shared memory mechanisms, which was however not considered in the scope of this work).

Although it is not at present possible to predict the performance of the implementation , some data can already be considered significant from the partitioning tool. The strategy implemented performs, in a first stage, a partition that is too fine-grained for our purposes. In a second stage, the sub-networks generated are regrouped in order to obtain a more desirable number of sub-networks. If this regrouping were not done, the performance would clearly be quite bad due to the overhead associated with dealing with a huge list of groups, each with only a few nodes. In that situation, almost all nodes will be "border" and probably 99 % will be either "border" or "nearborder" ; considering that only one of these nodes can be evaluated at a time, we have almost a uniprocessor version but an even slower one due to the overhead of determining which is the slowest group/node. This problem had already been noticed [9].

It becomes now evident also, that care must be taken however, because if the groups are too big, a lot of memory will be used in the checkpoints lists, and a lot of effort may be spent when rollback is needed. Clearly an optimal value will exist that maximizes the efficiency of the simulation control algorithm. This optimal value is definitely be dependent upon the circuit being simulated and its choice is certainly non-trivial. This considerations are of crucial importance in the results to be obtained.

References

[1] - Luís Vidigal et al.,

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

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

[2] - Luís Vidigal, Horácio Neto,

"CINNAMON: New Results and Improvements",

European Conference on Circuit Theory and Design, Paris, France, September, 1987

[3] - Kai Hwang, Fayé A.Briggs,

"Computer Architecture and Parallel Processing",

McGraw-Hill, 1984

[4] - Luís Miguel Silveira,

"Study of Multiprocessor Architectures for a Hardware Accelerator",

Esprit 1058 Report, January 1987

[5] - Luís Miguel Silveira,

"Multiprocessor Implementation of an event-driven Circuit Simulator: Evaluation of Design Options",

Esprit 1058 Report, July 1987

[6] - Alliant Computer Systems Corporation,

"Alliant FX Family"

[7] - Alliant Computer Systems Corporation,

"Concentrix C Handbook",

Part Number 302-00004-B, August 1986

[8] - José M. guimarães,

"MOS Transistor Models in CINNAMON",

Esprit 1058 Report, June 1987

[9] - Luís Miguel Silveira,

"A Static Partitioning Tool for Circuit Simulation",

Esprit 1058 Report, March 1987

[10] - Randal E. Bryant,

"An Algorithm for MOS Logic Simulation",

LAMBDA, 4^{th} Quarter, 1980

[11] - Jeffrey Arnold, Christopher J. Terman,

"A Multiprocessor Implementation of a Logic-Level Simulator",

International Conference on CAD, 1985