Parallel Circuit Simulation

An accelerated version of Cinnamon

Luís Miguel Silveira

Paulo Flores

INESC/IST

R. Alves Redol, 9

1000 Lisboa

PORTUGAL

Abstract

This report sumarizes the steps taken in the development of an accelerated version of the electrical circuit simulator CINNAMON. The architectural and functional features of an Alliant FX/4 computer are briely described and its use as the underlying hardware accelerator is outlined.

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.

This report summarizes the first attempt of an implementation of the underlying algorithm of **CINNAMON** 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. This important difference will, hopefully, be understood in the remaining chapters.

2 An Accelerated version of CINNAMON

Parallel Processing architectures and **algorithms** can be divided in three basic groups [3] [4]: **pipeline**,**array processing** and **multiprocessing**.

Pipelined architectures or algorithms derive from the generally called "Functional Partitioning Strategy" by means of which a particular set of operations is divided into related and consecutive steps that are performed sequentially on each set of data. Parallelism is achieved by having each stage of the pipeline performing a certain number of operations on the data. At each time instant, therefore, all the stages of the pipe can be active, performing operations on data that is later transferred to the next stage. The parallelism achieved by means of pipelining is usually described as performing a different set of operations on the same data "simultaneously".

Array Processing is a form of parallelism related with performing at the same time, the same operation on different data. In Array Processing machines the data is partitioned into a certain number of blocks and the same set of operations is simultaneously performed over all blocks. Array Processing is therefore usually referred as "Data Partitioning Parallelism".

Multiprocessing is a form of parallelism where on each processor a set of unrelated operations is performed over unconnected data. This structure is obviously the most versatile one, since all the others can be emulated by a multiprocessor machine although performance may decrease. This is sometimes the reason for its choice as the hardware architecture for certain implementations.

The results obtained from parallel implementations are highly dependent upon the application itself, i.e., the grain of parallelism achieved, the underlying hardware and the structure chosen for the implementation. In any architecture, the maximum theoretical speedup attainable is bounded by the number of processing units (stages in a pipeline or processing units). However that maximum is rarely attained due to the nature of the problem itself and the constraints it imposes.

In a pipelined architecture or one emulating a **Functional Partitioning** approach, maximum speedup is only achieved if at each time slot, the data being operated upon, at the various "stages", is not allowed to be "write-shared" (i.e., it cannot be written if any other stage of the pipe is using it), a characteristic that unfortunately is seldom found in practical algorithms. This is one reason why **Functional Parallelism** is not highly considered. Another reason is the difficulty to expand it to a great number of processing units ("stages"). Implementations derived from this strategy are, however, not particularly constrained by the architecture of the machine because the functional units are normally defined at high levels and make use of architectural or operating system primitives to emulate a pipelined environment.

Array Processors or **SIMD**, **Single Instruction on Multiple Data**, machines were traditionally special-purpose machines and therefore rarely considered as a parallel environment, due to its non-versatility. More recently, some effort has been devoted to its study and development and some machines of this kind have been used for many parallel computations (eg, the DAP machine and the Connection Machine).

Multiprocessors or **MIMD**, **Multiple Instruction on Multiple Data** machines are

the most widely used and explored architecture. They usually present features that allow the emulation of all other architectures, making them a very convenient environment for most developments.

From a certain point of view, this version can be understood as a data-partitioning implementation and it was developed on a multiprocessor machine [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.

2.1 The Hardware: Alliant FX/4

The Alliant machine installed at INESC is an FX/4 mini-supercomputer. 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, when the complex is formed, kept for a certain amount of time running parallel tasks, and then it is 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 to 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 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 mechanism supporting shared memory, usual forms of synchronization and atomic locks are also available.

The version of **CINNAMON** being described, was written 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:

concurrent_call(call_flag | 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(np, $arg_1, \ldots, arg_n$*)*

int np ;

...

{

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"* them 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:

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 (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 version of **CINNAMON** described in this report is built around concurrent

call's, using locks and unlocks as task synchronization primitives, wherever necessary.

2.2 General Concepts

CINNAMON is an event-driven electrical circuit simulator based on voltage axis discretization techniques. An event in **CINNAMON** 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.

In **CINNAMON**, 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 (time is continuous and there is no notion of time step in voltage discretization algorithms) the node's 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. The solution of the linear differential equation then, reduces to:

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

where $v_{k\infty} = i_k/g_k$ 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].

The approach taken in this version to achieve accelerated simulation will be described in the next chapter.

2.3 Simulation Control Algorithm

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

```
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 has to due 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() /* 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 ;
       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(proc,nd)           /* evaluation of perturbation
                                   propagation  */
int proc, nd ;
{
n =  node number proc attached to node 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 seldom used, was to detect the existence of computationally intensive portions of code that presented no data-dependency and having them run in parallel. 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 points were this technique was used.

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

3 Conclusions and Results

The basic trends for this accelerated version of **CINNAMON** have been described.

The results obtained so far can be considered as acceptable. The acceleration factor porsued has in fact been obtained. In the next pages, tables I and II present the results obtained for this version, for a series of benchmark circuits. Table I shows the total simulation time. Table II shows the timing results of the transient analysis of those simulations and is therefore a better measure of the results obtained.

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 for a National memory

Table I

Total Time (secs.)

Circuit	v4(VaxStation)	v4(Alliant)	v4(Accelerated)	speedup
100inv	1:32.33	21.82	11.66	1.87
2gi	15.97	4.33	3.69	1.17
2nc	51.27	14.41	11.94	1.21
a16	10:04.70	2:48.91	1:27.24	1.93
a32	11:24.95	3:02.47	1:48.15	1.69
a4	1:15.98	21.46	11.74	1.83
alu16	5:40.60	1:36.23	58.26	1.65
alu32	38:43.58	11:26.31	6:01.72	1.90
alu4	4:34.81	1:20.27	42.11	1.91
bi02	9:10.34	2:50.25	2:17.17	1.24
compt	16:21.55	4:25.81	2:18.35	1.92
fault	15.27	5.17	4.86	1.06
inv2	5.83	1.72	1.69	1.02
inv4	10.52	2.91	2.29	1.27
mem4	39.70	11.30	7.49	1.51
nand5	13.23	3.66	2.75	1.33
ring1	57.18	14.75	8.81	1.67
senser	21.3	5.67	4.03	1.41

Table II

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 ring oscillator. The other circuits are either too small or too active to present a considerable degree of parallelism.

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 access 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.

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 as been rewarding in many ways.

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