PARALLELIZING CIRCUIT SIMULATION - A COMBINED ALGORITHMIC AND SPECIALIZED HARDWARE APPROACH

Jacob White
IBM T.J. Watson Research Center

Nicholas Weiner
Department of Electrical Engineering and Computer Science
University of California, Berkeley.

ABSTRACT

Accurate performance estimation of high-density integrated circuits requires the kind of detailed numerical simulation performed in programs like ASTAP[1] and SPICE[2]. Because of the large computation time required for such programs when applied to large circuits, accelerating numerical simulation is an important problem. Parallel processing promises to be a viable approach to accelerating the simulation of large circuits. This paper presents an approach which exploits the parallelism in the simulation problem at two levels. A relaxation algorithm is used to break the circuit into loosely coupled blocks which can be computed in parallel, and special purpose hardware is used to exploit parallelism inside the block computation.

1. INTRODUCTION

Reliable and accurate simulation tools play a key role in Integrated Circuit (IC) design. This is because fabricating an integrated circuit is expensive and often time-consuming (on the order of months). In addition, minor errors in the integrated circuit design can not usually be corrected after fabrication. Therefore, design errors must be uncovered before fabrication, and this can be done through the use of simulation.

There are a wide variety of techniques for simulating integrated circuit designs, but none are as accurate, reliable, and technology independent as constructing the system of nonlinear ordinary differential equations that describe a given circuit, and solving this system with a numerical integration method. Specifically, implicit numerical integration algorithms are used to convert the differential equation system to a sequence of implicit nonlinear algebraic problems. The algebraic problems are solved using an iterative Newton method, which converts each nonlinear algebraic problem into a sequence of linear problems. This approach has been implemented in a variety of programs such as SPICE [1] and ASTAP[2].

The implicit integration methods are computationally expensive when applied to large systems because computing an implicit solution implies solving for all the variables in a given system simultaneously. However, the density of integrated circuit design has made large circuit simulation necessary. For this reason, several modifications to the direct numerical method above have been developed that avoid large system solution without compromising accuracy[3,4,5].

One of the techniques for numerically computing the solution to a large circuit that avoids performing a simultaneous solution of the entire system is the class of Waveform Relaxation algorithms[6,7]. WR algorithms are based on "lifting" the Gauss-Seidel and Gauss-Jacobi relaxation techniques for solving large algebraic systems to the problem of solving the large systems of ordinary differential equations. Briefly, the idea of WR is to first break a large circuit into a collection of subcircuits that are loosely intercoupled. Then the behavior of each subcircuit, over some interval of time, is calculated by "guessing" the behavior of the surrounding subcircuits over the same interval of time. The responses for each subcircuit are used to improve these guesses, and the response is recalculated. The procedure is iterated until the convergence is achieved for each block over the interval of time.

Not only are relaxation methods more efficient than direct methods for large problems, they are also promising techniques for parallel processors. The computational method decomposes a large circuit into a collection of small subcircuits, and these can be solved in parallel. And since the task of performing a subcircuit solution is a complicated computation, any parallel communication overhead is likely to be an insignificant portion of the computation time.

Performing the subcircuit computations in parallel, which will be referred to as *block-level* parallelism, has been the focus of most previous work[11,12,13,15]. However, it is only part of the parallelism that can be exploited in these methods. There is substantial parallelism inside the block computation, but it is more difficult to exploit because the computation is harder to decompose into independent pieces. In this paper, a two-level approach to parallelizing circuit simulation will be presented. As an introduction to exploiting block-level parallelism, previous work on a block parallel Waveform Relaxation (WR) algorithm is described. Then the block computation is detailed, and a special purpose parallel processor capable of exploiting the parallelism inside the block is presented.

2. A BLOCK-PARALLEL WAVEFORM RELAXATION ALGORITHM

The major difficulty in parallelizing the WR algorithm is that MOS digital circuits are highly directional, and it is important to follow that directionality when performing the relaxation computation, or the WR method will become inefficient. For example, consider applying WR to compute the transient response of a chain of inverters. If the first inverter's output is computed first, and the result is used to compute the second inverter's output, which is then used for the third inverter, etc., the resulting waveforms for this first iteration of the WR algorithm will be very close to the correct solution. However, if the second and third inverter outputs are computed in parallel with the first inverter's output, the results will not be close to the correct solution because no reasonable guess for the second and third inverter inputs will be available.

It is possible to parallelize the WR algorithm while still preserving a strict ordering of the computation of the subcircuit waveforms (Gauss-Seidel) by pipelining the waveform computation. In this approach, one processor would start computing the transient response to subcircuit. Once a first timepoint was generated, a second processor could begin computing the first timepoint for the second subcircuit, while the first processor computed the second timepoint for the first subcircuit. On the next step a third processor could be introduced, to compute the first timepoint for the third subcircuit, etc.

CH2348-1/86/0000/0438$01.00 © 1986 IEEE

One might suspect that this timepoint pipelining algorithm introduces too much scheduling overhead to be efficiently implemented on a loosely coupled parallel processor. The algorithm was implemented on a Sequent Balance 8000 system, a single-bus connected multiprocessor and the following table of results indicate that the algorithm is, in fact, effective[12].

TABLE 1 - TIMEPOINT PIPELINING WR CPU TIME VS # OF PROCESSORS

Circuit	FET's	1	3	6	9
uP Control	116	704	247	159	149
Eprom	348	745	265	185	182
Cmos Ram	428	3379	1217	642	496

3. INSIDE-BLOCK PARALLELISM

If a block relaxation method like WR is used, the problem of computing the time domain solution of a large circuit is reduced to computing the solutions to a sequence of smaller subcircuits. Computing a subcircuit's solution involves computing the solution to a system of the form

$$\dot{q}(v(t), u(t)) = f(v(t), u(t)) \quad v(0) = v_0 \qquad [1]$$

where $v(t) \in \mathbb{R}^n$ is the vector of node voltages, $u(t) \in \mathbb{R}^m$ is the vector of input voltages (either input sources to the large circuit, or node voltages from other subcircuits), $q(v(t), u(t)) \in \mathbb{R}^n$ is the vector of node charges, and $f(v(t),u(t))$ is the vector of net node currents. Here, n, the size of the subcircuit, will depend on the algorithm used to break the large circuit into subcircuits. Experience with the partitioning algorithms in RELAX2.3[11] indicate that an average n is near 10, and can vary from 1 to 100.

The trapezoidal numerical integration algorithm is usually used to solve the system of Eqn. 1. Given a timestep h, the trapezoidal integration algorithm applied to Eqn. 1 yields:

$$q(v(t+h),u(t+h)) - q(v(t),u(t)) - 0.5h(f(v(t+h),u(t+h)) + f(v(t),u(t))) = 0 \qquad [2]$$

where $v(t)$ is known, and the equation must be solved to compute $v(t+h)$. Equation 2 is solved with an iterative Newton-Raphson algorithm. The general Newton-Raphson iteration equation to solve $F(x) = 0$ is

$$J_F(x^k)\,(x^{k+1} - x^k) = -F(x^k) \qquad [3]$$

where $F(x^k)$ is referred to as the Newton residue and J_F is the jacobian matrix of F with respect to x. If the Newton algorithm is used to solve Eqn. 2 for $v(t+h)$, the residue, $F(v^k(t+h))$, is:

$$F(v^k(t+h)) = q(v^k(t+h),u(t+h)) - q(v(t),u(t)) - 0.5h(f(v^k(t+h),u(t+h)) + f(v(t),u(t))) \qquad [4]$$

and the Jacobian of $F(v^k(t+h))$, $J_F(v^k(t+h))$ is:

$$J_F(v^k(t+h)) = C(v^k(t+h),u(t+h)) + 0.5h\frac{\partial f(v^k(t+h),u(t+h))}{\partial v}. \qquad [5]$$

Then $v^{k+1}(t+h)$ is derived from $v^k(t+h)$ by solving the linear system of equations

$$J_F(v^k(t+h))\,[v^{k+1}(t+h) - v^k(t+h)] = -F(v^k(t+h)). \qquad [6]$$

The Newton iteration is continued until sufficient convergence is achieved, that is $\| v^{k+1}(t+h) - v^k(t+h) \| < \epsilon$ and $F(v^k(t+h))$ is close enough to zero.

The computation performed to calculate $v(t+h)$ from $v(t)$ is summarized in the following algorithm.

<u>**Algorithm 1 - (Computation of one timestep)**</u>

```
Pick a timestep $h$
Calculate the input vector $u(t + h)$
Calculate an initial guess for the Newton algorithm, $v^0(t + h)$.
repeat {
    for each ( element in the subcircuit ) {
        Compute the currents and the charges, and their derivatives.
        Sum the currents and charges into the Newton residue vector.
        Sum the derivatives into the Jacobian Matrix.
    }
    Decompose the Jacobian Matrix.
    Solve for the node voltage update.
    Check convergence
} until ( converged )
Calculate the integration error
```

There are many portions of the computation that can be parallelized[10,16]. Calculating the inputs, generating an intial guess, checking on the Newton convergence, and calculating the integration error are all normal vector operations and each element of the vector can be computed in parallel. However, the vector length will be short, because the subcircuit size will be near 10. The Jacobian decomposition and matrix solve can also be parallelized, but again the vectors will be short. The major portion of the computation, the device current, charge and derivative calculation, can be parallelized by computing all the devices in parallel. This calculation can be vectorized, but it is not possible to achieve peak efficiency because the calculations are data dependent[8]. Also, there will be memory contention when the devices add their contribution to the residue and the Jacobian. Since several devices can contribute to the same element of the residue vector or Jacobian entry, the summation must somehow be serialized.

4. SPECIAL HARDWARE TO EXPLOIT INSIDE-BLOCK PARALLELISM

As described above, there is substantial exploitable parallelism in the inside-block computation. However, inside-block parallelism is harder to exploit than the block parallelism because the computations that are performed in parallel involve many fewer operations. Interprocessor synchronization and communication will be performed much more frequently, and therefore must be very efficient, or any advantage provided by the parallelism will be lost.

The problem of designing a general-purpose parallel processor which allows for efficient synchronization and communication is extremely difficult. It is much easier to design a specialized system that will efficiently perform the parallel tasks involved in circuit simulation. In fact, considerably higher performance can be achieved by using a specialized design in which both the interconnection of processors and the design of the processors themselves are tuned to the circuit simulation task. This is the approach that has been adopted.

Consider the detailed profile of a 113-node memory buffer circuit simulated by the Relax2.3[11] program using the Yang-Chatterjee MOS model[18] (Table 2). As the table indicates, more than 96% of the time spent in the circuit simulation program is spent in the portion of the program represented by Algorithm 1. Since the rest of the computer time is spent performing input and output processing, the approach taken to simulation acceleration is to construct a specialized co-processor for an existing general purpose host computer. The general purpose computer then performs all the input and output, and the instruction set of the co-processor is limited to only those instructions needed to perform the steps in Algorithm 1.

TABLE 2 - CPU USAGE IN A SAMPLE SIMULATION	
Input/Output	1.98s
Timestep selection	0.01s
Element evaluation, Jacobian and Residue load	42.8s
Matrix Decomposition	5.66s
Matrix Solve and Node Voltage update	5.63s
Testing Newton Convergence	1.43s
Local Truncation Error Calculation	0.86s

4.1 - Basic Coprocessor Architecture

The are several ways the co-processor design is tuned to performing the steps in Algorithm 1. Since almost all the operations are performed on double precision floating point numbers, the data path to memory is very wide (128 bits). As pointed out above, since the size of typical subcircuits will be about 10 nodes, the co-processor is set up as an five-way parallel processor, so that each of the processors will be used efficiently. And because the WR algorithm reduces large problems to computing a sequence of small problems, the memory space of the processor is limited (512k bytes). The small memory requirement makes it possible to make the entire memory very fast. This is a tremendous advantage because it eliminates the need for a cache. In addition, since computing each Newton iteration involves touching every datum in the problem, a fast memory is the only way to insure rapid execution[17].

The co-processor is constructed from a five-stage pipelined processor. Using a deeply pipelined processor as a single serial machine does not generally provide peak performance. This is because in most programs, branches and interinstruction data dependencies frequently disrupt the pipeline. To avoid this, the pipeline processor is designed so that each stage of the pipeline executes an independent instruction stream, so the co-processor can be treated as a five-way parallel processor with a single shared memory[14]. In a sense, the intractable problem of eliminating interinstruction data dependencies and branches has been transformed to the standard parallel synchronization problem. Note that it would be simpler to design the unit as a vector processor, but the bulk of the computation, element current and charge evaluation, can not be efficiently vectorized.

The entire 512k byte co-processor memory is shared not only by the five sub-processors, but also by the host computer, allowing for easy access to results and eliminating the need to copy data. The host and the co-processor take turns performing operations on a single copy of the data in shared memory.

4.2 - Specialized Instruction Set.

Most of the operations performed in Algorithm 1 are double precision (64 bit) floating point operations. For this reason the co-processor instruction set includes mostly floating point instructions. The co-processor cycle time is set by the floating point instruction time, so that a floating point instruction executes in one cycle.

The path to memory is 128 bits wide to allow each double precesion number to be associated with a 32 bit pointer and a 32 bit integer. Since all three are read each time a double precision operand is used, simultaneous operations can performed on the three fields. This allows the co-processor to step through a linked list in a single instruction loop, which is useful for performing operations on vectors and sparse matrices[19].

Most analytic transistor models use complex functions like logarithm, exponential and square root. Standard algorithms for computing these functions can be accelerated through the use of tables. For this reason, the co-processor includes a special table-lookup facility. It is possible to use the facility for both standard mathematical function evaluation as well as user-defined functions.

In order to efficiently deal with the problem of synchronization, the processor also has a double precision serial add function. Using the serial add function, each of the five processors can safely add quantites to identical locations in memory, and the result will be the same as if the processors performed the additions in some strict order. This is particularly useful when forming the residue and the Jacobian (see above).

4.3 - Task Distribution Mechanism

The pipelined co-processor is viewed as five parallel sub-processors. Each has its own program counter, and operates in the usual fashion, executing the instruction indicated by its program counter and incrementing the program counter. Program flow is controlled by the use of branch instructions.

A mechanism, involving very little host processor - co-processor communication, for distributing tasks amongst the sub-processors has been devised. The mechanism involves the use of a "higher level" sequence of instructions, or "task queue", within co-processor instruction memory, and of an extra program counter to keep track of progress through this sequence. Each of the instructions in the sequence is a branch to a routine to perform one of the processing tasks. Every time a sub-processor completes a task it picks up the next instruction from the higher level sequence, and commences execution of the next processing task. This procedure is continued until all of the processing tasks have been completed.

5. CONCLUSIONS AND AKNOWLEDGEMENTS

The cycle time of the co-processor hardware is 0.24 microseconds, and since floating point operations take one cycle, the co-processor's peak rate is 4.17 megaflops. Several of the routines in the RELAX2.3[11] program that are used in Algorithm 1 have been translated into the co-processor's assembly language. Examination of generated assembly code indicates that the co-processor will achieve between 2 and 4 megaflops for most of the computations involved in Algorithm 1. It is difficult to project the final system performance, because it will depend crutially on the interaction between the host and co-processor. A detailed register-transfer level simulation study is in progress, and results will be presented at the conference.

The authors would like to acknowledge the architecture suggestions of Jack Kohn and Monty Denneau, and the many valuable discussions with Giovanni DeMicheli, Dan Ostapko, and Albert Ruehli.

REFERENCES

[1] W. T. Weeks, A. J. Jimenez, G. W. Mahoney, D. Mehta, H. Qassemzadeh and T. R. Scott, "Algorithms for ASTAP -- A Network Analysis Program," *IEEE Trans. on Circuit Theory,* Vol. CT-20, pp. 628-634, Nov. 1973

[2] L.W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Electronics Research Laboratory *Rep. No. ERL-M520, University of California, Berkeley, May 1975.*

[3] B.R. Chawla, H.K. Gummel, and P. Kozah, "MOTIS - an MOS Timing Simulator," *IEEE Trans. Circuits and Systems, Vol. 22, pp. 901-909, 1975*

[4] K. Sakallah and S.W. Director, "An Activity-Directed Circuit Simulation Algorithm," *Proc. IEEE Int. Conf. on Circ. and Computers,* October 1980, pp.1032-1035

[5] R. A. Saleh, J. E. Kleckner and A. R. Newton, "Iterated Timing Analysis and SPLICE1", *ICCAD'83 Digest,* Santa Clara, CA., 1983.

[6] E. Lelarasmee, A. E. Ruehli, A. L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits," *IEEE Trans. on CAD of IC and Syst.,* Vol. 1, n. 3, pp.131-145, July 1982.

[7] P. Defebve, J. Beetem, W. Donath, H.Y. Hsieh, F. Odeh, A.E. Ruehli, P.K. Wolff, Sr., and J. White, "A Large-Scale Mosfet Circuit Analyzer Based on Waveform Relaxation" *International Conference on Computer Design,* Rye, New York, October 1984.

[8] A. Vladimirescu and D. O. Pederson, "A Computer Program for the Simulation of Large Scale Integrated Circuits," *IEEE Proc. Int. Symp. on Circuits and Systems,* Chicago, 1981.

[9] J. T. Deutsch "Algorithms and Architecture for Multiprocessor-Based Circuit Simulation", *Ph.D. Dissertation,* University of California, Berkeley, Electronics Research Laboratory, 1985

[10] R. Ginosar and N. G. Jacobson "The Simulation Machine: A VLSI Architecture for Circuit Simulation" *Proc. Int. Conf. on Comp. Design,* Port Chester, New York, October 1985

[11] J. White and A.L. Sangiovanni-Vincentelli, "Partitioning Algorithms and Parallel Implementations of Waveform Relaxation Algorithms for Circuit Simulation" *Proc. Int. Symp. on Circ. and Syst.,* Kyoto, Japan, June 1985

[12] J. White, R. Saleh, A. Sangiovanni-Vincentelli, and A. R. Newton, "Accelerating Relaxation Algorithms for Circuit Simulation using Waveform Newton, Iterative Step Size Refinement, and Parallel Techniques" *Int. Conf. on Computer-Aided Design,* Santa Clara, California, November 1985.

[13] H. Uno et al., "A Parallel Implementation of MOS Digital Circuit Simulation", *Int. Conf. on Computer-Aided Design,* Santa Clara, California, November 1985.

[14] R. S. Gyurcsik and D. O. Pederson, "A MOS Transistor Model-Evaluation Attached Processor for Circuit Simulation", *Proc. of the Int. Conf. on Computer-Aided Design,* Santa Clara, California, November 1985.

[15] An Empirical Analysis of the Performance of a Multiprocessor-Based Circuit Simulator. G.K. Jacob, A.R. Newton, D. O. Pederson *Proc. of the Design Automation Conference,* Las Vegas, Nevada, June 1986.

[16] C. L. Seitz, "The Cosmic-Cube", *Comm. of the ACM,* Jan. 1985, pp. 22-33.

[17] D. Webber, *Private Communications,* June, 1985

[18] P. Yang, B.D. Epler, P. K. Chatterjee, "An Investigation of the charge conservation problem for MOSFET circuit simulation," *IEEE Journal of Solid-State Circuits,* Vol. SC-18, No. 1, pp. 128-138, Feb. 1983

[19] K. S. Kundert, "Sparse Matrix Techniques", A. E. Ruehli (editor), *Circuit Analysis, Simulation and Design. Vol. 1.,* North-Holland, to be published in 1986.