FA6 - 9:00

Proceedings of 24th Conference
on Decision and Control
Ft. Lauderdale, FL • December 1985

WAVEFORM RELAXATION TECHNIQUES AND THEIR PARALLEL IMPLEMENTATION

Alberto L. Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences

University of California at Berkeley, CA 94720

Jacob White

I.B.M. Watson Research Center

Yorktown Heights, New York, 10598

Summary. Because of the high cost of fabricating an Integrated Circuit(IC), it is important to verify the design using simulation. There are a wide variety of techniques for simulating integrated circuit designs, but the most accurate is to construct the system of nonlinear ordinary differential equations that describe a given circuit, and solve the system with a numerical integration method. This approach, referred to as circuit simulation, is computationally expensive, particularly when applied to large circuits. To reduce the computation time required to simulate large MOS circuits, new numerical integration algorithms based on relaxation techniques have been developed. These techniques can reduce the simulation time as much as an order of magnitudes over standard circuit simulation programs. In addition, they are particularly suited for parallel implementation. In this paper we will focus on the Waveform Relaxation (WR) family of algorithms. Algorithms in this family will be reviewed, convergence theorems will be offered, and their implementation on a parallel processor presented.

1. Introduction

Reliable and accurate simulation tools must 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. This approach, referred to as circuit simulation, has been implemented in a variety of programs such as SPICE [NAG75]. These programs use a standard, or direct, techniques based on the following four steps:

i) An extended form of the nodal analysis technique to construct a system of the differential equation system from the circuit topology.

ii) Stiffly stable implicit integration methods, such as the Backward Difference formulas, to convert the differential equations which describe the system into a sequence of nonlinear algebraic equations.

iii) Modified Newton methods to solve the algebraic equations by solving a sequence of linear problems.

iv) Sparse Gaussian Elimination to solve the systems of linear equations generated by the Newton method.

Circuit simulation tools based on the above techniques are heavily used. Companies spend many millions of dollars per year in computer costs, and a number of companies run over 60,000 simulations/month. However, these programs were designed in the early 1970's for the simulation of circuits with a few hundred transistors at most. They are now being applied, somewhat inappropriately, to the task of simulating digital and analog VLSI circuits, which can contain more than 50,000 devices. As problems increase in size, it becomes less economically feasible to use the above direct techniques. SPICE [NAG75] can take several hours (on a VAX11/780) to simulate circuits with only a few hundred devices.

There are two reasons why the direct approach described above can become inefficient for large systems. The most obvious reason is that sparse matrix solution time will grow super-linearly with the size of the problem. Experimental evidence indicates that the point where the matrix solution time begins to dominate is when the system has over several thousand nodes, and this is the size of systems that are beginning to be simulated for state of the art IC designs.

The direct methods become inefficient for large problems also because for large differential equation systems, the different state variables are changing at very different rates. Direct application of the integration method forces every differential equation in the system to be discretized identically, and this discretization must be fine enough so that the fastest changing state variable in the system is accurately represented. If it were possible to pick different discretization points, or time-steps, for each differential equation in the system so that each could use the largest time-step that would accurately reflect the behavior of its associated state variable, then the efficiency of the simulation would be greatly improved.

Several modifications of the direct method have been used that both avoid large sparse matrix solutions, and allow the individual equations of the system to use different time-steps [CHA75, NEW78, GEA80, SAK80, DEM80, LEL82, NEW83, SAL83, WHI84, CHI84]. One class of such techniques, Waveform Relaxation [LEL82, WHI84, WHI85a, WHI85b], is 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 associated with MOS digital circuits. Briefly, the idea of these relaxation technique is to first break a large circuit into loosely coupled subcircuits. 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 subcircuit over the interval of time. Other relaxation techniques such as the Gauss-Seidel-Newton algorithm [ORT70] can be applied to solve the nonlinear system of algebraic equations in place of the standard Newton-Raphson techniques.

Two circuit simulation programs have been developed at Berkeley using relaxation techniques: RELAX, based on Waveform Relaxation[LEL82, WHI84, NEW83] and SPLICE, based on Iterated Timing Analysis (ITA) [NEW83, SAL83], a form of Gauss-Seidel-Newton technique. On a uniprocessor, these programs can show speed improvements over direct methods of up to an order of magnitude even for problems with only a few hundred devices. In addition, both the ITA and Waveform Relaxation are particularly amenable to use on multiprocessors because the computational method already decomposes the problem. A distributed form of the ITA algorithm, called DITA, has been recently developed and a prototype DITA simulator, the MSPLICE program, has been implemented[DEU84].

In this paper, we review the basic Waveform Relaxation algorithms and their numerical properties. In addition, we examine the issues involved in the implementation of these algorithms on parallel processors. In particular, Section 2 is dedicated to the basic Waveform Relaxation algorithms and their convergence properties. In Section 3, we present numerical techniques that make the basic algorithms particularly efficient for the analysis of large scale circuits. In Section 4, two parallel algorithms are presented and the architecture used for the implementation of the algorithms is discussed. In Section 5, concluding remarks and future directions are discussed.

2. Waveform Relaxation Algorithms

We will start this section with a simple illustrative example, and then present the general WR algorithm. Consider the first-order two-dimensional differential equation in: $x(t) \in \mathbb{R}^2$ on $t \in [0,T]$.

$$\dot{x}_1 = f_1(x_1, x_2, t) \qquad x_1(0) = x_{10} \tag{2.1a}$$

$$\dot{x}_2 = f_2(x_1, x_2, t) \qquad x_2(0) = x_{20} \tag{2.1b}$$

The basic idea of the waveform-relaxation algorithm is to fix the waveform $x_2: [0,T] \rightarrow \mathbb{R}$ and solve Eqn. (2.1a) as a one dimensional differential equation in $x_1(t)$. The solution thus obtained for $x_1(t)$ can be substituted into Eqn. (2.1b) which will then reduce to another first-order differential equation in one variable, $x_2(t)$. Eqn. (2.1a) is then re-solved using the new solution for $x_2(t)$ and the procedure is repeated.

Alternately, fix the waveform $x_2(t)$ in Eqn. (2.1a) and fix $x_1(t)$ in Eqn. (2.1b) and solve both one dimensional differential equations simultaneously. Use the solution obtained for x_2 in Eqn. (2.1b) and the solution obtained for x_1 in Eqn. (2.1a) and re-solve both equations.

In this fashion, iterative algorithms have been constructed. Either replaces the problem of solving a differential equation in two variables by one of solving a sequence of differential equations in one variable. As described above, these two waveform relaxation algorithms can been seen as the analogues of the Gauss-Seidel and the Gauss-Jacobi techniques for solving nonlinear algebraic equations. Here, however, the unknowns are waveforms (elements of a function space), rather than real variables. In this sense, the algorithms are techniques for time-domain decoupling of differential equations.

The most general formulation of a system of nonlinear differential equations is the following implicit formulation:

$$F(\dot{x}(t), x(t), u(t)) = 0 \qquad x(0) = x_0 \tag{2.2}$$

where $x(t) \in \mathbb{R}^n$ on $t \in [0,T]$; $u(t) \in \mathbb{R}^r$ on $t \in [0,T]$ is piecewise continuous; and $F: \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^r \rightarrow \mathbb{R}^n$ is continuous.

In order to guarantee that WR applied to Eqn. (2.2) will converge to the system's solution, we first must guarantee that Eqn. (2.2) has a solution. If we require that there exists a transformation of Eqn. (2.2) to the form $\dot{y} = f(y,u)$ where f is Lipschitz continuous with respect to y for all u, then a unique solution for the system exists[HAL69]. Although there are many sets of broad constraints on F that guarantee the existence of such a transformation, the conditions can be difficult to verify in practice. In addition, for the above system, it is difficult to determine how to assign variables to equations when applying the WR algorithm. That is, when solving the F_i equation of the system in the iteration proces , what x_j variable should be solved for implicitly. If a poor choice is made, the relaxation may not converge[LEL82].

Rather than carefully considering the existence and assignment questions, which will complicate the analysis that follows without lending much insight, we will consider the following less general form, in which many practical problems, particularly circuit simulation, can be described.

$$C(x(t), u(t))\,\dot{x}(t) = f(x(t), u(t)) \qquad x(0) = x_0 \tag{2.3}$$

where $x(t) \in \mathbb{R}^n$ on $t \in [0,T]$; $u(t) \in \mathbb{R}^r$ on $t \in [0,T]$ is piecewise continuous; $C: \mathbb{R}^n \times \mathbb{R}^r \rightarrow \mathbb{R}^{n \times n}$ is such that $C(x,u)^{-1}$ exists and is uniformly bounded with respect to x,u; and $f: \mathbb{R}^n \times \mathbb{R}^r \rightarrow \mathbb{R}^n$ is globally Lipschitz continuous with respect to x for all $u(t) \in \mathbb{R}^r$.

The fact that $C(x,u)$ has a well-behaved inverse guarantees the existence of a normal form for Eqn.(2.3), and that $x(t) \in \mathbb{R}^n$ is the vector of state variables for the system. Then as f is globally Lipschitz with respect to x for all u, $C(x,u)^{-1}$ is uniformly bounded, and $u(t)$ is piecewise continuous, there exists a unique solution to Eqn. (2.3).

The WR algorithm for solving the above system is as follows:

Algorithm 2.1 (WR Gauss-Seidel Algorithm for solving Eqn. (2.3))

Comment:
The superscript k denotes the iteration count,
the subscript i denotes the component index
of a vector and ϵ is a small positive number.

$k \leftarrow 0$;
guess waveform $x^0(t)$; $t \in [0,T]$
such that $x^0(0) = x_0$
(for example, set $x^0(t) = x_0$, $t \in [0,T]$);

repeat {

$k \leftarrow k+1$

foreach ($i \in \{1,..,n\}$) {

solve

$$\sum_{j=1}^{i} C_{ij}(x_1^k, \cdots, x_i^k, x_{i+1}^{k-1}, \cdots, x_n^{k-1}, u)\dot{x}_j^k + \sum_{j=i+1}^{n} C_{ij}(x_1^k, \cdots, x_i^k, x_{i+1}^{k-1}, \cdots, x_n^{k-1}, u)\dot{x}_j^{k-1} - f_i(x_1^k, \cdots, x_i^k, x_{i+1}^{k-1}, \cdots, x_n^{k-1}, u) = 0$$

for ($x_i^k(t)$; $t \in [0,T]$), with the initial condition $x_i^k(0) = x_{i_0}$.

}

} until ($\max_{1 \le i \le n} \max_{t \in [0,T]} |x_i^k(t) - x_i^{k-1}(t)| \le \epsilon$)

that is, until the iteration converges.

Note that the differential equation in Algorithm 2.1 has only one unknown variable x_i^k. The variables $x_{i+1}^{k-1}, \cdots, x_n^{k-1}$ are known from the previous iteration and the variables $x_1^k, \cdots, x_{i-1}^k$ have already been computed. Also, the Gauss-Jacobi version of the WR Algorithm for Eqn. (2.3) can be obtained from Algorithm 2.1 by replacing the foreach statement with the forall statement and adjusting the iteration indices.

Under rather mild conditions, in general satisfied by circuits of interest, a global convergence theorem can be proven.

Theorem 2.1 [WHI85a,LEL82] If in addition to the conditions on $C(x,u)$ and f listed above, the matrix $C(x,u)$ is strictly diagonally dominant uniformly over all x and u and Lipschitz continuous with respect to x for all u then the sequence of waveforms $\{x^k\}$ generated by the Gauss-Seidel and the Gauss-Jacobi versions of Algorithm 2.1 converge uniformly to the solution of Eqn. (2.3) for all bounded intervals $[0,T]$ for all initial guesses that satisfy the initial conditions.

This convergence theorem can be proven by first showing that if $C(x,u)$ is diagonally dominant, then there exists a bound on the $\dot{x}^k$'s generated by the WR algorithm that is independent of k. Using this bound, it can be shown that the assumption that $C(x,u)$ is Lipschitz continuous implies there exists a norm on $\mathbb{R}^n$ such that for arbitrary positive integers j and k:

$$\|\dot{x}^{k+1}(t) - \dot{x}^{j+1}(t)\| \le \gamma\|\dot{x}^k(t) - \dot{x}^j(t)\| + l_1\|x^{k+1}(t) - x^{j+1}(t)\| + l_2\|x^k(t) - x^j(t)\|$$

for some $\gamma < 1$ and $l_1, l_2 < \infty$ for all $t \in [0,T]$. In the properly chosen norm $\|\cdot\|_b$ on $C([0,T],\mathbb{R}^n)$, the above equation implies that

$$\|\dot{x}^{k+1} - \dot{x}^{j+1}\|_b < \|\dot{x}^k - \dot{x}^j\|_b$$

and therefore the sequence $\{\dot{x}^k\}$ converges by the contraction mapping theorem. As $x^k(0) = x_0$ for all k, $\{x^k\}$ converges as well.

3. Techniques to Speed-up WR Algorithms

In this section, we analyze several of the implementation techniques used to improve the efficiency of the basic WR algorithm, and give theorems that indicate the strengths or limitations of these techniques. We begin by considering breaking the simulation interval into pieces, called windows, and demonstrate that the technique can be used to reduce the number of relaxation iterations required to achieve satisfactory convergence. We then examine how to partition large systems into loosely coupled subsystems. Finally, we present an ordering algorithm that accelerates WR convergence by exploiting the directional nature of the Gauss-Seidel relaxtion algorithm.

3.1. Windowing

Consider the following nonlinear ordinary differential equation in $x_1(t), x_2(t) \in \mathbb{R}$ with input $u \in \mathbb{R}$ that approximately describes the cross-coupled *nor* logic gate in Fig. 3.1.1a (the approximate equations represent a normalization that converts the simulation interval $[0,T]$ to $[0,1]$).

$$\dot{x}_1 = (5 - x_1) - x_1(x_2)^2 - 5x_1u \tag{3.1.1}$$

$$\dot{x}_2 = (5 - x_2) - x_2(x_1)^2$$

$$x_1(0) = 5.0 \qquad x_2(0) = 0.0$$

The Gauss-Seidel WR Algorithm given in Section 2 was used to solve for the behavior of the cross-coupled *nor* gate circuit approximated by the above small system of equations. In Fig. 3.1.1b plots of the input $u(t)$, the exact solution for $x_1(t)$, and the relaxation iteration waveforms for $x_1(t)$ for the 5th, 10th and 20th iterations are shown. The plots demonstrate a property typical of the WR algorithm when applied to systems with strong coupling: the difference between the iteration waveforms and correct solution is not reduced at every time point in the waveform. Instead, each iteration lengthens the interval of time, starting from zero, for which the waveform is close to the exact solution.

As an example of "better" convergence, consider the following differential equation in x_1, x_2, x_3 with input u that approximately describes the shift register in Fig. 3.1.2a (here the simulation interval $[0,T]$ has been normalized to $[0,1]$)

$$\dot{x}_1 = (5.0 - x_1) - x_1(u)^2 - (x_1 - x_2) \tag{3.1.2}$$

$$\dot{x}_2 = (x_1 - x_2)$$

$$\dot{x}_3 = (5.0 - x_3) - x_3(x_2)^2$$

$$x(0) = 0.$$

The Gauss-Seidel WR Algorithm given in Section 2 was used to solve the original system approximated by the above system of equations. The input $u(t)$, the exact solution for $x_1(t)$, and the waveforms for $x_1(t)$ computed from the first, second, and third iterations of the WR algorithm are plotted in Fig. 3.1.2b. As the plots for this example show, the difference between the iteration waveforms and the correct solution is reduced throughout the entire waveform.

Perhaps surprisingly, the behavior of the first example is consistent with the WR convergence theorem, even though that theorem states that the iterations converge uniformly. This is because it was proved that the WR method is a contraction map in the following nonuniform norm on $C([0,T],\mathbb{R}^n)$:

$$\max_{[0,T]} e^{-bt} \| f(t) \|$$

where $b > 0$, $f(t) \in \mathbb{R}^n$, and $\| \bullet \|$ is a norm on $\mathbb{R}^n$. Note that $\| f(t) \|$ can increase as e^{bt} without increasing the value of this function space norm. If $f(t)$ grows slowly, or is bounded, it is possible to reduce the function space norm by reducing $\| f(t) \|$ only on some small interval in $[0,T]$, though it will be necessary to increase this interval to decrease further the function space norm. The waveforms in the more slowly converging example above, converge in just this way; the function space norm is decreased after every iteration of the WR algorithm because significant errors are reduced over larger and larger intervals of time. The examples above lead to the following definition:

Definition 3.1.1: If the WR algorithm is used to solve a given a differential system of the form given in Eqn. (2.3), then *WR uniform iteration factor* for the differential system is defined as the smallest $\gamma > 0$ such that

$$\max_{[0,T]} \| x^{k+1}(t) - x^k(t) \| \leq \gamma \max_{[0,T]} \| x^k(t) - x^{k-1}(t) \|$$

where $x(t) \in \mathbb{R}^n$ on $t \in [0,T]$ is the solution to Eqn. (2.3); $x^k(t) \in \mathbb{R}^n$ on $t \in [0,T]$ is the k^{th} iterate of Algorithm 1; and $\| \bullet \|$ is any norm. Futhermore, the differential system is said to have the *strict WR contractivity property* on $[0,T]$, if the WR algorithm applied to the system is a contraction map in a uniform norm on $[0,T]$, i.e. for some norm on $\mathbb{R}^n$, $\gamma < 1$. If the WR algorithm applied to the system is a contraction in a uniform norm on $[0,T]$ for any $T > 0$ then we say that the system has the strict WR contractivity property on $[0,\infty)$.

For a system of equations to have the strict WR contractivity property on $[0,\infty)$ it must be more than just loosely coupled. In addition, the decomposed equations solved at each iteration of the waveform relaxation must be well-damped, so that errors due to the decomposition die off in time, instead of accumulating or growing. As the crossed *nand* gate example indicates, many systems of interest do not have the strict WR contractivity property on $[0,T)$ for all $T < \infty$. However, we can prove that any system that satisfies the WR convergence theorem will also have the strict WR contractivity property on some nonzero interval.

Theorem 3.1.1: For any system of the form of Eqn. (2.3) which satisfies the assumptions of the WR convergence theorem (Theorem 2.1) there exists a $T > 0$ such that the system has the strict WR contractivity property on $[0,T]$.

Theorem 3.1.1 guarantees that the WR algorithm will be a contraction mapping in a uniform norm for any system, provided the interval of time over which the waveforms are computed is made small enough. This suggest that the interval of simulation $[0,T]$ should be broken up into *windows* $[0,T_1], [T_1,T_2], \ldots, [T_{n-1},T_n]$ where the size of each window is small enough so that the WR algorithm contracts uniformly throughout the entire window. The smaller the window is made, the faster the convergence. However, as the window size becomes smaller, the advantages of the waveform relaxation are lost. Scheduling overhead increases when the windows become smaller, since each subsystem must be processed at each iteration in every window. If the windows are made very small, time-steps chosen to calculate the waveforms will be limited by the window size rather than by the local truncation error, and unnecessary calculations will be performed.

The lower bound for the region over which WR contracts uniformly given in Theorem 3.1.1 is too conservative in most cases to be of direct practical use. As mentioned above, in order for the WR algorithm to be efficient it is important to pick the largest windows over which the iterations actually contract uniformly, but the theorem only provides a worst-case estimate. Since it is difficult to determine *a priori* a reasonable window size to use for a given nonlinear problem, window sizes are usually determined dynamically, by monitoring the computed iterations[WHI84]. Since Theorem 2.1 guarantees the convergence of WR over *any* finite interval, a dynamic scheme does not have to pick the window sizes very accurately. The only cost of a bad choice of window is loss of efficiency, the relaxation will *still* converge.

3.2. Partitioning

In the basic WR algorithm presented above, the node equations are solved as single differential equations in one unknown, and these solutions are iterated until convergence. This kind of node-by-node decomposition can lead to slow convergence in the case where a few nodes in a large system are tightly coupled. For example, consider the circuit in Fig. 3.2.1, a simple three resistor, two capacitor circuit. If the floating resistor connecting the two nodes is large relative to the grounded resistors, then the WR algorithm using node-by-node decomposition converges rapidly. However, if the floating resistance is small, then the convergence will be very slow.

For this reason, the first step in almost every WR-based program is to partition the system, that is, to scan all the nodes in the system and determine which should be lumped together and solved directly. Partitioning well is difficult for several reasons. If too many nodes are lumped together, the advantages of using relaxation will be lost, but if any tightly coupled nodes are not lumped together then the WR algorithm will converge very slowly. And since the aim of WR is to perform the simulation rapidly, it is important that the partitioning step not be computationally burdensome.

Several approaches have been applied to this partitioning problem. One simple approach is to require the user to partition the system [WHI83, DEF84, CAR84]. This technique is reasonable for the simulation of large circuits because large circuits usually have already been broken up into small, fairly independent pieces to make the design easier to understand and manage. Unfortunately, what is a sensible partitioning from a design point of view may not be a good partitioning for the WR algorithm. Another approach to partitioning is to examine the topology of the circuit to find functional blocks (i.e. a nand gate or a flip-flop)[LEL82b]. The nodes of the system are then grouped together based on being a member of a functional block. This type of partitioning is difficult to perform, since the algorithm must recognize broad classes of functional blocks, and non-standard blocks may not be treated properly.

Since it is the intent of the partitioning to improve the speed of convergence of the relaxation, it is sensible to partition a large circuit with this, rather than topology or functionality, in mind. In addition, techniques that examine only the topology of the circuit can not, in general, partition the circuit in such a way that the relaxation converges rapidly. It is essential to examine the numerical value of the components. We have developed algorithms based on this idea[WHI85b].

As it is difficult to get estimates of the speed of WR convergence directly, a relationship will be shown between the convergence speed of the WR and that of two simpler problems. We will then present techniques for estimating the speed of convergence for the simpler problems, and show how they are used to perform partitioning.

The WR uniform iteration factor γ defined in the previous section is a lower bound on how fast the iteration waveforms approach the exact solution. A good partitioning algorithm will keep this γ small without generating unnecessarily large subcircuits.

As mentioned at the beginning of this section, γ is difficult to estimate directly for a given problem. However, we have the following theorems which relate this γ to iteration factors for reduced algebraic problems.

Theorem 3.2.1: Let γ be the WR uniform iteration factor for a given system of equations of the form of Eqn. (2.3) solved on [0,T], and assume that all v,u such that $f(v,u) = 0$ are points of attraction for the differential equation system. Then in the limit as $T \to \infty$, γ is bounded below by the iteration factor for solving the reduced problem $f(v,u)=0$, for $v \in \mathbb{R}^n$ given any $u \in \mathbb{R}^n$.

Theorem 3.2.2: Let γ be the WR uniform iteration factor for a given a system of equations of the form of Eqn. (2.3). Then γ is bounded below by the relaxation iteration factor for solving the reduced problem $C(y,u)v = b$, for $v \in \mathbb{R}^n$ given any $y,u,b \in \mathbb{R}^n$.

The proofs of these theorems are straight-forward, and will not be presented here.

Since in Eqn. (231) $C(v,u)$ is the matrix of linear and nonlinear capacitors, and $f(v,u)$ is the net circuit currents generated by conductances, the two theorems above indicate that it is possible to get lower bound estimates of γ by examining the capacitances and conductances independently. And since these estimates are lower bounds, if they are then used to drive a partitioning algorithm, the partitioner will not generate unnecessarily large subcircuits. That is, in order to decrease γ, the WR uniform iteration factor, it is *necessary* to partition in such a way that the iteration factors for the reduced problems are decreased.

In order to develop an algorithm for estimating the iteration factor for the conductance problem, we will start with a simple problem for which we can calculate the iteration factor exactly, and then apply this result, by analogy, to larger problems. Consider a simple three resistor circuit derived by removing the capacitors from the circuit in Fig. 3.2.1. If Gauss-Seidel relaxation is used to solve this two node problem then the iteration equations can be written down by inspection as:

$$v_1^{k+1} = \frac{g_3}{(g_1+g_3)} v_2^k$$

$$v_2^{k+1} = \frac{g_3}{(g_2+g_3)} v_1^{k+1}$$

The iteration factor can be computed exactly and is:

$$\gamma = \frac{g_3}{(g_2+g_3)} \frac{g_3}{(g_1+g_3)}$$

At this point we introduce a heuristic; the circuit in Fig. 3.2.1 can be used as a simple model for the coupling between two nodes in a MOS circuit. With this model the MOS transistor will be treated as a nonlinear resistor from its source to its drain. Coupling between the gate and source and gate and drain will be considered in the next section on ordering. With this simplification, we can use the following algorithm for partitioning circuits with two-terminal resistances.

Algorithm 3.1 (Conductance Partitioning)

```
for each ( conductive element in the circuit ) {
        g3 ← maximum element conductance over all v.
        Remove the element from the circuit.
        Replace the rest of the conductances in the circuit
                by their minimum values over all v.
    Compute g1 and g2, the Norton Equivalent
        conductances to ground at the two element terminals.

    Comment:
        α is user-supplied, and is set to be the largest
        acceptable contraction factor. Typical value is 0.3.

    If ( g3/(g2+g3)  g3/(g1+g3) > α ) {
        Tie the two terminal nodes together.
    }
}
```

The Norton equivalent conductances at a node can be approximated using a simple recursive formula [WHI85b].

Unfortunately, it is difficult to relate the partitioning criteria α to the eventual iteration factor for the WR, or even the iteration factor for the conductance problem. This relationship can be determined only for special cases.

Since the capacitance problem is almost identical in nature to the conductance problem, the capacitance partitioning algorithm follows almost the same strategy as the conductance partitioning. The major difference is that instead of comparing floating capacitances to Norton equivalent conductances, they are compared to equivalent capacitances. These equivalent capacitances are entirely analogous to the equivalent conductances, and can be computed with the same recursive approach used for the conductances.

Along with applying the partitioning algorithm to a variety of MOS digital circuits, we have also applied this partitioning algorithm to a large VHSIC memory circuit with 2900 nodes and over 3500 parasitic components. The results matched our own best attempts to partition the circuit in as many instances as we had the patience to check. However, we still suspect that if the method is applied to larger problems, the subcircuits produced may become quite large. Should this be the case, we plan to extend the present algorithm by performing an additional pass over only the excessively large subcircuits, and subpartitioning them using more sophisticated algorithms. In particular, to use better estimates of the equivalent conductances and capacitances, as we feel the present algorithm may be unnecessarily conservative.

In addition, the partitioning algorithm presented here, is static, i.e., the circuit is partitioned only once at the beginning of the simulation by using the worst case values of the devices. It is not frequent that all the devices attain their worst case values at the same time. It would be more effective to partition the circuit dynamically, i.e., to update the subcircuits during the analysis. The problem with this approach is overhead since the cost of repartitioning the circuit is not negligible. We have developed a dynamic partitioning scheme for bipolar circuits where a worst case analysis would very often end up with very large subcircuits. We plan to extend this algorithm to MOS circuits.

3.3. Ordering

When applying the Gauss-Seidel WR algorithm to a decomposed system of differential equations, the order in which the equations are solved can strongly effect the number of WR iterations required to achieve satisfactory convergence. In order to explain this effect, consider the case where there are only grounded two-terminal capacitors for each node of the circuit. Thus, the matrix $C(x,u)$ of Eqn. (2.2) is diagonal. Then let the *dependency matrix* of f in Eqn. (2.2) be defined as a zero-one matrix $P = [p_{ij}]$ such that $p_{ij} = 1$ if f_i depends on x_j, $p_{ij} = 0$ otherwise. Note that P also represents the zero-nonzero structure of the Jacobian of f.

If P is lower triangular, then one iteration of the Gauss-Seidel WR algorithm will produce the exact solution to the original differential equation system (in practice, two iterations will be performed because a second iteration is needed to verify that convergence has been achieved). If P is not lower triangular, but the dependency of the f_i component of f on x_j, $i<j$, is "weak", then the result of one iteration of the Gauss-Seidel WR algorithm will be close to the exact solution, and subsequent iterations will converge rapidly. For this reason, when applying relaxation techniques to the solution of circuit equations, the technique can be made much more efficient by reordering the equations to make P as close to a lower triangular matrix as possible.

As discussed in Section 3.1, subsets of nodes in a large circuit may be mutually tightly coupled, and in order to insure that the relaxation algorithm converges rapidly when applied to such a circuit, these subsets are grouped together into subcircuits and solved with direct methods. This corresponds to *block* relaxation method, and an ordering algorithm applied to a system being solved with block relaxation should attempt to make f as *block lower triangular* as possible.

In some sense, partitioning and ordering the subsystem of equations are performing similar functions. They are both attempting to eliminate slow relaxation convergence due to two nodes in a large circuit being tightly coupled. There is, however, a key difference. If, for example, x_i is strongly dependent on x_j, and x_j is strongly dependent on x_i, then the partitioning algorithm will lump the two nodes together into one subcircuit. However, if x_i is strongly dependent on x_j, but x_j is *weakly* dependent on x_i, then the two nodes will not be lumped together, but the ordering algorithm should insure that the system is block lower triangular by ordering the equations so that x_j is computed before computing x_i.

Resistors and capacitors do not exhibit the kind of unidirectional coupling that is of concern to the ordering algorithm. In fact, the only element type of concern to the ordering algorithm are transistors, because they exhibit unidirectional coupling. That is, the drain and source terminals of an MOS transistor are strongly dependent on the gate terminal of the transistor, but the gate is almost independent of the drain and source. Clearly, this implies that the subcircuits containing the given transistor's drain or source should be analyzed before the subcircuit containing the given transistor's gate.

To devise an algorithm to carry out this task, it is convenient to introduce the *dependency* graph of the partitioned circuit. If we represent the circuit with a directed graph $G(X,E)$, where the set of nodes, X, is in one-to-one correspondence with the subcircuits obtained by a partitioner, and where there is a directed edge between the node corresponding to subcircuit i and the node corresponding to subcircuit j if there is a transistor whose gate is in subcircuit i and whose drain or source are in subcircuit j. If the graph is acyclic, it can be levelized, i.e. all the nodes can be ordered in *levels* so that a node in level i can have incoming edges only from nodes in levels lower than i. The ordering so obtained is the one used by RELAX2.3 to process the subcircuits.

However, there may be cases where cycles exist in the graph. In this case, the ordering algorithm either changes the subcircuit definitions by grouping two or more subcircuits together, effectively performing part of the partition task, or it discards edges to remove the cycles. In both cases, at the end of this process we obtain an acyclic graph and an ordering of the subcircuits corresponding to the leveling of the (perhaps altered) graph.

One question remains, which is when to repartition to remove a feedback loop versus breaking the loop. As the example in the section on windowing indicates, if signal propagation around the feedback loop is fast compared to the size of the window, slow nonuniform convergence will result. For this reason, the ordering algorithm makes the decision about partitioning based on an estimate of the delay around the feedback loop. If it is smaller than one percent of the simulation interval, the feedback loop is removed by repartitioning. If the delay is larger, then the feedback loop is broken by removing an edge from the directed graph.

Algorithm 3.3.1 (Relax2.3 Subcircuit Ordering Algorithm)

```
/* Initialization. */
ordered_list = NULL;
unordered_list = List of subcircuits from the partitioner;
/* Main Loop. */
while (unordered_list != NULL) {
  none_ordered == FALSE;
  while (none_ordered == FALSE) {
    none_ordered == TRUE;
    for each (subcircuit in the unordered_list) {
      if (all subcircuits on incoming arcs are on ordered_list) {
        none_ordered = FALSE;
        append_to_end_of_ordered_list(subcircuit);
        delete_from_unordered_list(subcircuit);
      }
    }
  }
  if (unorder_list != NULL) {
  /* Must be a feedback loop. */
    found_loop = FALSE;
    depth = 1;
    while (found_loop == FALSE) {
      depth = depth + 1;
      for each (subcircuit in the unordered list) {
        if
        (there exists a loop of length = depth) {
          found_loop == TRUE;
          if (delay around the loop >
          0.01 * the simulation interval) {
            break the loop;
          }
```

```
            else {
              collapse loop into one subcircuit;
            }
          }
        }
      }
    }
  }
```

4. Parallel Implementation of WR Algorithms

Exploiting parallel computation for circuit simulation is extremely important because the size of the circuits for which circuit simulation has been applied has grown at rate that far exceeds the increase in computational power due to technological improvement. The *only* way to keep pace with the increasing demand is to be able to apply many processors to the problem, and the number of processors that can be used must scale up with the size of the problem.

In this section, the implementation of two WR-based parallel circuit simulation algorithms on a shared-memory computer will be described. We start by presenting a brief overview of the aspects of a shared-memory computer that effect the algorithm implementation, and then describe the two parallel WR algorithms, one based on using a mixture of Gauss-Seidel and Gauss-Jacobi relaxation, and the other based on pipelining the waveform computation.

4.1. The Shared-Memory Computer

To write efficient programs for serial computers, knowledge about the specific details of the architecture is useful, but not essential. This is not the case for programming on a parallel computer. Specific details about the architecture can influence decisions about the implementation of an algorithm, and can even affect the choice of algorithm.

The WR parallel algorithms have been implemented on the Sequent Balance 8000, a shared-memory parallel computer. In this subsection we describe those aspects of the architecture that affected this implementation.

The key problem in designing a parallel processor is that of communication between the processors. One simple approach is to design a parallel computer by gathering together many standard serial computers, and connecting them together with a communication network. Usually such computers are referred to as *message-passing* parallel computers, because data is transferred between the many processors by passing messages on the communication network. The disadvantage of such a system is that in order to move data from the memory of one processor into the memory of the second processor, both the transmitting and receiving processors must be involved.

Another approach to the problem of communicating between parallel processors is to redesign the memory system, so that aggregate memory of all the processors is directly addressable by any one of the individual processors. Such a system is referred to as a *shared-memory* system because the processors are all sharing a single resource, the memory. The main advantages of a shared-memory machine is that it is not necessary to explicitly transfer data from one processor to another. When a processor needs data from another processor, it simply reads from the memory locations in which the other processor has written. This also allows for more dynamic algorithm structures, because it is not necessary to determine beforehand which processors will need the results of a given calculation. The disadvantages of the shared-memory computer are that any processor can destroy the contents of all memory, and guaranteed synchronization between processors is not easy without special purpose hardware. We will discuss the synchronization issue further below.

There are fundamentally two choices for memory organization in a shared memory multi-processor. Either the entire memory is centralized and all the processors contend for access to it or the memory is distributed so that each processor "owns" a portion of the shared memory, i.e., a processor can access its memory more quickly than it can access memory owned by other processors.

The two different organizations can impact the structure of parallel algorithms. In fact, if the memory is distributed among the processors, a parallel algorithm performs better if the data for the computation can be partitioned so that each processor uses only the data in its own portion of the shared memory. On the other hand, partitioning the data according to the criterion above may cause some of the parallelism of a given algorithm to be lost.

The memory on the Sequence Balance 8000 is centralized, i.e., all the processors contend for one large shared memory. For such an architecture, there is no advantage to partitioning the data among the processors, as they will still have to contend for the same centralized memory pool. Because of the memory contention problem, most implementations of shared-memory computers that use a centralized memory include a large cache memory with each of the processors. The cache mechanism is intended to exploit *reference locality*. In other words, it is assumed that each of the processors is actively using and reusing only a small amount of data. As with standard caches, the processor caches buffer the most recently used data requested from the central memory. Thus, it is expected that each of the processor finds its data in the cache and need not access the central memory.

Using caches on a parallel computer is not as straight-forward as on a serial computer. Each of the many caches is supposed to maintain a copy of a part of the data stored in central memory. Because any of the processors can write in any memory location at any time, it is possible for the caches to loose *consistency* i.e., the contents of a cache may not reflect the current contents of the central memory. For example, if the contents of memory location A is in the cache for processor 2 and processor 1 updates A, then the data in the cache for processor 2 will be incorrect.

There are a variety of schemes for avoiding this problem, but we will only mention the technique applied in the computer used for experimentation. The scheme is simple, all the caches monitor all the writes to central memory from any of the processors. If a cache contains the location being written to, it updates its own copy of the data in the given location. In this way, it is assured that the data in all the caches is current.

This cache architecture can be exploited to avoid excessive memory traffic when many processors are waiting for a processor to complete a computation and to write the result in the central memory. The processors could continuously interrogate the central memory to check whether the processor has written its data. However, in this way, excess memory traffic is created and the computing processor has to slow down. If the cache architecture described above is used, the excess traffic is eliminated. Each waiting processor keeps reading a location which is in its own cache, and therefore does not generate any central memory traffic. When the computing processor finishes, each of the other processor caches will spot the write to the monitored location in central memory and each cache will update its own copy of the data. The waiting processors will therefore be made immediately aware of the completion of the computing processor, but will not have impeded the progress of the computing processor by generating excess memory traffic.

The last aspect of the parallel computer architecture that we consider is that of mutual exclusion or *locking*. In almost all parallel programs there are critical sections that must be performed serially. When a critical section is executed, only one processor must be active. To make sure that no other processor is executing that section, a *test-and-set* instruction is used.

If a processor executes a *test-and-set* instruction on a given location in memory, the contents of the location is returned to the processor and simultaneously, if the location was not set, it is set. This mechanism can be used to perform locking as follows. A particular location in memory is used as the lock. If a processor is about to execute a critical section of a parallel program, it first executes a test-and-set on the lock location. If the result of the test-and-set indicates that the location was previously unset, the processor has the lock. If not, the processor must wait until the lock location becomes free, and then retry the test-and-set. Once a processor has the lock, it can safely executes the critical section as no two processors can have the lock at the same time. Once the processor finishes the critical section, it clears the lock location and then other processors can get the lock and execute the critical section.

4.2. The Gauss-Seidel-Jacobi Algorithm

An obvious way of parallelizing WR is to apply the Gauss-Jacobi version of WR. In this algorithm, the relaxation makes use of the waveforms computed at the previous iteration for all the subsystems. Then all the subsystems could be analyzed independently by different processors. One of the difficulties in applying this algorithm is that MOS digital circuits are highly directional, and, if this directionality is not exploited, slow convergence may result. 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 inverters' 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 inverters' inputs will be available.

Following a strict ordering of the relaxation computation (Gauss-Seidel) does not allow for computing entire waveforms in parallel, and computing the next iteration waveforms for every subcircuit at once (Gauss-Jacobi) allows for substantial parallelism, but is not very efficient (converges more slowly). To preserve the efficiency of the Gauss-Seidel algorithm and allow for some of the parallelism of Gauss-Jacobi, a mixed approach must be employed.

Examining large digital circuits, we have observed that the dependency between inputs and outputs of the subcircuits tend to be "wide", i.e., there are several "parallel" chains of gates, with some interaction among chains. It is then possible to order the computation so that subcircuits in

parallel "chains" can be computed in parallel, but the serial dependence inside a chain is preserved. This will not allow for as much parallelism as the Gauss-Jacobi scheme, but should preserve most of the efficiency of the Gauss-Seidel scheme.

In Algorithm 4.1, we present an approach that follows as much as possible the ordering used in the Gauss-Seidel WR algorithm implemented in RELAX2.3. In this algorithm, subcircuits are ordered according to the levelling of the dependency graph. In particular, no subcircuit at level i is analyzed before all the subcircuits at previous levels have been examined. Of course, subcircuits at the same level can all be processed in parallel without affecting substantially the convergence of the algorithm (it may affect slightly the convergence because of weak coupling between subcircuits at the same level that is not represented in the dependency graph). If we have several processors and few subcircuits at a particular level, then, if we wait for the level to be fully analyzed before moving to the next level, we may have many idle processors. We address this problem by letting the processors analyze subcircuits at the next levels before the present level is fully examined. Because the analysis of these subcircuits in the Gauss-Seidel algorithm needs the waveforms of the subcircuits at previous levels that may not be available, the waveforms computed at the previous iteration are used. This scheme introduces some Gauss-Jacobi "steps" in a general Gauss-Seidel scheme. We force all the processors to synchronize at the end of the iteration, i.e. if there are no subcircuits left to analyze in the ordered list (actually the data structure used for this ordered list is a queue), then the processors that finish their jobs are left idle until all the processors end up being idle. At that point, the queue is reinitialized and all the processors "grab" a subcircuit from the queue. This strategy makes sure that the steps taken inside an iteration are either Gauss-Seidel or Gauss-Jacobi steps. Note that the global convergence properties of WR algorithms still hold, the speed of convergence may be slower though when several Gauss-Jacobi "steps" are taken by the algorithm. These steps are fewer if the number of subcircuits at any level is substantially larger than the number of processors. We cannot predict a priori how many Gauss-Jacobi "steps" will be taken, because this depends on the time spent in processing the various subcircuits. This time is a random variable due to the many unpredictable factors affecting the computation.

The attributes of the parallel architecture of the machine used for our experiments have been considered in the algorithm. First, the data describing the subcircuits and the computed waveforms are stored in the central shared memory, to be accessed as needed. Note that when a processor is free and wants to process a circuit, it has to access the queue to find a subcircuit waiting to be analyzed. If two processors access the queue at the same time, we may have that the same subcircuit is processed by two processors. Thus, whenever a processor is picking up a subcircuit from the queue, it locks the queue so that no other processor can access it until he has finished. At that point, the active processor unlocks the queue and the other processors can access the queue. Each of the processors waits for the queue to be free by examining the lock variable in a tight loop. As mentioned above, this exploits the nature of the cache consistency strategy. Finally, in this case it is not necessary to separately control access to the waveforms. Since the waveforms will only be written as a result of the computations performed on their associated subcircuits, and a waveform is associated with only one subcircuit, the mutual exclusion of the subcircuit queue will prevent writes from colliding.

Algorithm 4.1 (Jacobi/Seidel Based Parallel WR)

```
/* Initialization. Both subcircuits and waveforms in shared-memory. */
    queue = ordered_list_of_subcircuits;

/* Parallel iteration loop. All processors execute. */
   while (all_converged == FALSE) {
       if (processor == 1) {
           reset_queue();
           idle_count = 0;
       }
       /* If idle_count != # of processors, some processor is still computing. */
       while(idle_count != number of processors) {
           /* Tight loop waiting for queue to unlock. */
           while(test-and-set(queuelock) == set) {};
           /* Queue is locked, get next subcircuit */
           NextSub = Get_next_queue_entry();
           if(NextSub == NULL) {
               increment(idle_count);
               clear(queuelock);
           } else {
           /* There is another subcircuit on the queue. */
               clear(queuelock);
               Compute_Subcircuit_Waveforms(NextSub);
               Check_Waveform_Convergence(NextSub);
           }
       }
   }
```

4.3. The Time-point Pipelining Algorithm

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 for a subcircuit. Once a first time-point is generated, a second processor could begin computing the first time-point for the second subcircuit, while the first processor computes the second time-point for the first subcircuit. On the next step a third processor could be introduced, to compute the first time-point for the third subcircuit, and so on.

Conceptually, the operations of a given processor in a parallel time-point pipelining algorithm are quite simple. The algorithm is set up by establishing both the space in shared memory for storage of the iteration waveforms, and a buffer or queue with the list of subcircuits. Each of the processors then starts by taking a subcircuit from the queue. The individual processors examine their respective subcircuit's external waveforms to see if the waveform values needed to compute the next integration time-step are available. If so, the next time-step for the subcircuit is computed. Otherwise, the subcircuit is returned to the queue and the processor tries again with another subcircuit from the queue. As time-points are computed, more of the subcircuits will have the information needed to compute their own time-points.

As one might expect, a practical time-point pipelining algorithm is more complicated that the conceptual algorithm. Perhaps the most obvious difficulty is that there is a tremendous overhead in having every processor search through all the subcircuits to find one of the few for which a time-point can be computed. It is possible to reduce the number of candidate subcircuits a processor must search by only considering those subcircuits for which at least one of the external waveforms has more time-points than it had when the subcircuit was last processed. Clearly this will avoid having the processors continuously rechecking subcircuits for which no new information is available, and therefore no new time-step could be computed.

This kind of selective search algorithm can be implemented by altering the way the queue of subcircuits is used. When a processor discerns that it is not possible to compute a new time-point for a subcircuit, instead of returning the subcircuit to the queue, the subcircuit is temporarily discarded. If a processor succeeds in computing a time-point for a subcircuit, those subcircuits that are connected to the given subcircuit, referred to as the *fanouts* of the subcircuit, are added to the queue (Of course, the fanouts that are already on the queue are not duplicated). In this way, the only subcircuits that will be on the queue are those for which it is likely that the waveform values needed to compute a next time-point will be available.

Another aspect of the time-point pipelining algorithm that increases the exploitable parallelism at the cost of slightly complicating the algorithm is to allow the time-point pipelining to extend across iteration boundaries. For example, consider a chain of two inverters, and assume that it takes two time-steps to compute each of the inverter outputs. As before, the second time-step of the first inverter can be computed in parallel with the computation of the first time-step of the second inverter. Then, while the second time-step of the the second inverter is being computed, there is enough information to compute the first time-step of the first inverter for the *second* WR iteration.

This enhancement doesn't really complicate the conceptual algorithm, until one considers the question of when to stop. For a long chain of inverters, allowing the pipelining to extend across iteration boundaries can easily allow for the first inverter to be many iterations ahead of the last inverter. Since WR convergence can only be determined when all the waveforms for a given iteration have been computed, it may well be that the WR iteration being computed for the first inverter is many iterations beyond what is necessary to achieve satisfactory convergence. The difficulty is that this fact will not be discovered until much later, when all inverter outputs have been computed for the iteration for which satisfactory convergence is achieved.

In this case, the algorithm will still produce correct solutions, but unnecessary computations will be performed and efficiency will be degraded. The unnecessary computations are reasonably simple to avoid, by not allowing any subcircuit to start on iteration $N+1$ until nonconvergence of some waveform of iteration N has been detected. It is, of course, important to discover as quickly as possible if it will be necessary to compute iteration $N+1$ so that the pipelining of that iteration can begin. For this reason, in the time-point pipelining algorithm presented below, convergence is checked on a time-point by time-point basis, immediately after a time-point is computed.

Algorithm 4.2 (Time-point Pipelining WR Algorithm)

```
/* Initialization. Both subcircuits and waveforms in shared-memory. */
  queue = ordered_list_of_subcircuits;
  idle_count = 0;
  /*
    Max_iter_so_far indicates the iteration after the last one for which
```

```
    nonconvergence has been detected.
  */
  max_iter_so_far = 1;
/* Parallel iteration loop. All processors execute. */
  /* If idle_count != # of processors, some processor is still computing. */
  while (idle_count != number of processors) {
    /* Tight loop waiting for queue to unlock. */
    while (test-and-set(queuelock) == set) {};
    /*
      Queue is locked, get next subcircuit in the queue on which the
      work that might be performed is for an iteration that is no more
      than one beyond the maximum iteration for which nonconvergence
      has been detected.
    */
    NextSub = Get_next_queue_entry(max_iter_so_far);
    if (NextSub == NULL) {
      increment(idle_count);
      clear(queuelock);
    }
    else {
    /*
      There is another subcircuit on the queue whose iteration is not
      beyond max_iter_so_far.
    */
      clear(queuelock);
      /*
        Compute as many time-points as possible with available
        waveform values.
      */
      repeat {
        /*
          Check to see if external values needed to compute the next
          time-step are available.
        */
        cando = Check_for_next_step(NextSub);
        if (cando == TRUE) {
          Compute_Next_Step(NextSub);
          converged = Check_Step_Convergence(NextSub);
          /*
            Keep max_iter_so_far ahead of the nonconverged iterations.
          */
          if ((converged == FALSE) and (NextSub.iteration_count ==
          max_iter_so_far) {
          increment(max_iter_so_far);
        }
        enqueue_fanouts(NextSub);
      }
      until (cando == FALSE)
    }
  }
```

Relaxation-based algorithms are finding their way in the industrial community. Much work remains to be done to improve the speed of the algorithms. In particular, more sophisticated partitioning algorithms should be devised. The use of multi-processors and special purpose hardware for circuit simulation has attracted the attention of many researchers. Our results are preliminary. We are carrying out experiments on a variety of different architectures to investigate the relationships between algorithms and computer architecture. In particular, new algorithms will be studied for the hypercube architecture of the Intel iPSC and of a massive parallel computer under development at International Thinking Machines.

The work on relaxation-based simulation has been carried out in collaboration with a number of colleagues and students. It is our pleasure to mention the interaction with Prof. Richard Newton and his students, in particular Res Saleh. Some of the numerical analysis aspects of Waveform Relaxation have been developed in collaboration with F. Odeh of IBM. Discussions with Prof. W. Kahan are gratefully acknowledged. Ken Kundert and Peter Moore have contributed to the development of RELAX2.3. Guy Marong has developed a version of RELAX for bipolar circuits. Donald Webber has implemented a relaxation-based simulator on the International Thinking Machine massive parallel computer. Finally, we would like to thank Shiva Multisystems for the use of their equipment and facilities, and Sequent Computers for the use of their multiprocessor system.

This research has been sponsored by DARPA under contract NESC-N39, IBM, Philips Research Labs, and a grant from the MICRO program of the State of California.

Table 4.1 - Gauss-Seidel/Gauss-Jacobi WR on several # of Processors.

	Circuit FET's	1	3	6	9
uP Control	66	595	338	270	259
Eprom	348	512	317	286	266

Table 4.2 - Time-point Pipelining WR Algorithm on several # of Processors.

	Circuit FET's	1	3	6	9	VAX/780
uP Control	116	704	247	159	149	240
Eprom	348	745	265	185	182	212
CMOS Ram	428	3379	1217	642	496	960

4.4. Experimental Results

As mentioned above, the two algorithms were implemented on a 9 processor configuration of the Sequent Balance 8000 computer (larger configurations are available) The results from several experiments for the two algorithms are given in Tables 4.1 and 4.2. As the results from the Eprom and microprocessor control circuit indicate, the time-point pipelining algorithm makes much more efficient use of the available processors. In fact, as Table 4.2 shows, the time-point pipelining algorithm running on the Balance 8000 runs substantially faster than the serial WR algorithm running on a VAX/780.

A second point should be made about the time-point pipelining examples. The speed-up does not remain linear to nine processors, but starts to drop off after seven processors. This is surprising, given the size of the examples, but not when the type of circuit being simulated is considered. For the biggest example, the CMOS RAM, the partitioning algorithm produces approximately 75 subcircuits, and this would indicate that a speed-up of 75 should be obtainable, or at least approachable. However, this reasoning ignores one of the features of the WR algorithm: only those portions of the circuit that are active are processed. For digital circuits, usually less than ten percent of the circuit is active. This implies that, for the CMOS RAM example over any given interval, roughly seven subcircuits are active and, hence, involved in the computation. Therefore only a speed-up of seven could be expected.

5. Conclusions

We have presented an overview of Waveform Relaxation algorithms and of their numerical properties. In addition, techniques to speed up the execution of the algorithms have been introduced. The major contribution of this paper is in the presentation and discussion of parallel WR algorithms. Two algorithms have been investigated and their performance on a Sequent Balance 8000 multi-processor have been given.

References

[CAR84] C. H. Carlin and A. Vachoux, "On Partitioning for Waveform Relaxation Time-Domain Analysis of VLSI Circuits" *Proc. 1984 Int. Symp. on Circ. and Syst.*, Montreal, Canada, May 1984.

[CHA75] 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*

[CHE84] C. F. Chen and P. Subramaniam, "The Second Generation MOTIS Timing Simulator-- An Efficient and Accurate Approach for General MOS Circuits" *Proc. 1984 Int. Symp. on Circ. and Syst.*, Montreal, Canada, May 1984.

[DEF84] 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.

[DEM80] G. De Micheli, A. Sangiovanni-Vincentelli and A.R. Newton, "New Algorithms for the Timing Analysis of Large Circuits" *Proc. 1980 Int. Symp. on Circ. and Syst.*, Houston, 1980.

[DEU85] J. T. Deutsch "Algorithms and Architecture for Multiprocessor-Based Circuit Simulation", Ph.D. Dissertation, University of California, Berkeley, Electronics Research Laboratory, 1985

[GEA80] C. William Gear, "Automatic Multirate Methods for Ordinary Differential Equations" *Information Processing 80*, North-Holland Pub. Co., 1980

[HAL69] J. K. Hale, *Ordinary Differential Equations* John Wiley and Sons, Inc., 1969

[LEL82a] 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.

[LEL82b] E. Lelarasmee and A. Sangiovanni-Vincentelli, "Relax: a new circuit simulator for large scale MOS integrated circuits", Proc. 19th Design Automation Conference, Las Vegas, Nevada, pp. 682-690, June 1982.

[NAG75] L.W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Electronics Research Laboratory *Rep. No. ERL-M520, University of California, Berkeley, May 1975.*

[NEW78] A. R. Newton, "The Simulation of Large Scale Integrated Circuits", *Memorandum UCB/ERL M78/52*, July 1978.

[NEW83] A.R. Newton and A. L. Sangiovanni-Vincentelli "Relaxation-Based Circuit Simulation" *IEEE Trans. on ED*, Vol. ED-30, N. 9, pp. 1184-1207, Sept. 1983 also *SIAM Jour. on Scientific and Stat. Computing*, Vol. 4, N. 3, Sept. 1983 also *IEEE Trans. on CAD of IC and Syst.*, July 1984.

[ORT70] J. M. Ortega and W.C Rheinbolt, *Iterative Solution of Nonlinear Equations in Several Variables* Academic Press, 1970.

[SAK80] 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

[SAL83] R. A. Saleh, J. E. Kleckner and A. R. Newton, "Iterated Timing Analysis and SPLICE1", *ICCAD'83 Digest*, Santa Clara, CA., 1983.

[WHI83] J. White and A. L. Sangiovanni-Vincentelli, "RELAX2: A Modified Waveform Relaxation Approach to the Simluation of MOS Digital Circuits" *Conf. Proc. IEEE ISCAS*, Vol. 2, pp756-759, Newport Beach, CA, May, 1983.

[WHI84] J. White and A. Sangiovanni-Vincentelli, "Relax2.1 - A Waveform Relaxation Based Circuit Simulation Program" *Proc. 1984 Int. Custom Integrated Circuits Conference* Rochester, New York, June 1984.

[WHI85a] J. White, F. Odeh, A. Sangiovanni-Vincentelli, A. Ruehli, "Waveform Relaxation - Theory and Practice" *Trans. on Computer Simulation*. To appear.

[WHI85b] 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, ne 1985