

Parallel Waveform-Newton Algorithms for Circuit Simulation

Eugene Z. Xia and Resve A. Saleh, *Member, IEEE*

Abstract—This paper describes the parallelization of the waveform-relaxation-Newton (WRN) method for circuit simulation. The techniques are applied at three levels of granularity: subcircuits are processed in parallel for large-grain parallelism, time points are processed in parallel for medium-grain parallelism, and devices are processed in parallel for fine-grain parallelism. The effectiveness of each of these approaches is demonstrated with the PSPLAX program (which is a parallel version of a WRN program called SPLAX) using a number of industrial examples. Currently, PSPLAX is implemented on an Alliant FX/80 with eight processors. The results provided in this paper indicate that the combination of these approaches will also be effective on larger systems.

I. INTRODUCTION

OVER the past few years, parallel processors have been used in circuit simulation to reduce run times in the analysis of large digital circuits. Both the direct methods used in the SPICE2 program [1] and the relaxation methods [2] in programs such as RELAX [3] and SPLICE [4] have been parallelized. While the attempts to parallelize the techniques in SPICE2 have been successful (see, for example, [5]–[10]), the relaxation-based algorithms appear to be better suited to parallel VLSI circuit simulation owing to the recognized limits in parallelizing the sparse linear equation solution process [10]–[14]. As a result, a significant amount of work has also been performed in parallelizing waveform relaxation (WR) [15]–[18] and iterated timing analysis (ITA) [19], [20]. Recently, the waveform-relaxation-Newton (WRN) algorithm [15], [21] has been proposed as an effective relaxation-based method for moderately coupled multirate problems on uniprocessors. This paper describes a variety of techniques to perform WRN in parallel. The implementation described here is for the Alliant FX/80 multiprocessor [22], which is a shared-memory machine with eight processors. However, the algorithms appear to be very well suited to systems with more than eight processors, as will be demonstrated in this paper.

Manuscript received June 12, 1990. This work was supported by the National Science Foundation (NSF MIP-8410110), by the Department of Energy (DOE-DE-FG02-85ER25001), by the Air Force Office of Scientific Research (AFOSR-85-0211), and by a donation from IBM. This paper was recommended by Associate Editor A. E. Ruehli.

E. Z. Xia was with the Center for Supercomputing Research and Development, University of Illinois, Urbana, IL. He is now with the University of Maryland, College Park, MD 20742.

R. A. Saleh is with the Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL 61801.

IEEE Log Number 9105734.

Waveform-Relaxation-Newton Algorithm

The WRN algorithm was shown to be very effective on a special class of circuits, namely large, moderately coupled, multirate, digital MOS circuits [21]. The differential equations describing such a system are usually specified in the following way:

$$\dot{q}(v(t)) = -f(v(t), u(t)), \quad v(0) = V, \quad t \in [0, T], \quad (1)$$

where q is the sum of the charges arising from the capacitors connected to each node, f is the sum of the currents charging the capacitors at each node, $v(t) \in \mathbb{R}^n$ is the set of unknown node voltages, and $u(t)$ is the set of input source voltages. The circuit simulation problem is to solve for the unknown node voltages as a function of time in the simulation interval $t \in [0, T]$ given a set of initial conditions specified by the vector V .

An algorithmic description of WRN to solve the above system and the time-step control used in conjunction with it are given below. The basic idea is to linearize and solve each differential equation in the inner loop of the waveform relaxation method. The motivation and details of the WRN algorithm and its time-step control have been covered in a recent paper [21] and will not be repeated here. In presenting the complete simulation algorithm using WRN, the following notation will be used:

$$v^{k,i}(t) = [v_1^k(t), \dots, v_{i-1}^k(t), v_i^{k-1}(t), \dots, v_n^{k-1}(t)]_T.$$

Algorithm 1: Gauss-Seidel WRN Algorithm

$k \leftarrow 0$;
guess waveform $v^0(t)$; $t \in [0, T]$ such that $v^0(0) = v_0$;
repeat {

$k \leftarrow k + 1$;
foreach ($i \in \{1, \dots, n\}$) {
 solve

$$\frac{\partial}{\partial t} \left[q_i(v^{k,i}(t)) + \frac{\partial q(v^{k,i}(t))}{\partial v} (v_i^k(t) - v_i^{k-1}(t)) \right] -$$

$$f_i(v^{k,i}(t)) + \frac{\partial f_i(v^{k,i}(t))}{\partial v_i} (v_i^k(t) - v_i^{k-1}(t)) = 0$$

for ($v_i^k(t)$; $t \in [0, T]$), with the initial condition $v_i^k(0) = v_{i0}$;

```

    }
  }
  until (max1 ≤ i ≤ n maxt ∈ [0, T] || vik(t) - vik-1(t) || ≤ ε)
  ■

```

The equation solved in the inner loop of Algorithm 1 is the original differential equation linearized by the waveform-Newton method [23], which is then solved by standard integration methods to find the waveform at iteration k . As in iterated timing analysis (ITA) [4], the solution of each nonlinear differential equation is approximated using only one step of the waveform-Newton algorithm, thereby decreasing the per-iteration cost of this WR-based approach.

A special time point selecting mechanism was developed to take advantage of certain features of the waveform-relaxation-Newton algorithm [15], [21]. During the first iteration, a very large step is taken which is equal to the interval of integration, called a window [3]. When an approximate solution to the system is obtained, the local truncation error is estimated at all the solution points to determine the time points for the next iteration. If the local truncation error is too large, the mechanism subdivides the interval into two steps and selects two mesh points for the second iteration. When the approximate solutions for the second iteration are obtained, another check of the local truncation errors at the two solution points is made to determine whether more mesh points are required in the next iteration. If the local truncation error for the first mesh point is too large, another mesh point is placed in the middle of the first subinterval. On the other hand, if the local truncation error for the second mesh point is small, the mechanism does not add any more points to that subinterval. This process is repeated in subsequent iterations until the waveforms converge. The key point here is that the time points are known prior to performing each iteration in a given window.

The algorithm and time-step control described above have been implemented in the SPLAX program and the results of simulations using this program have been reported in [21]. Within SPLAX, and other WR-based programs, there are three major factors which affect the overall performance: circuit partitioning, subcircuit ordering after partitioning, and window size selection [3]. Of these, the window size selection is perhaps the most difficult to perform optimally. If the window is too large, convergence may be slow for circuits with tight coupling and large feedback loops. On the other hand, if the window is too small, then it is difficult to exploit latency and multirate behavior efficiently. Therefore, choosing the optimal window size is critical for achieving the highest performance. In our approach, the window size also plays a key role in the efficiency of the parallel code, as will be seen.

II. LARGE-GRAIN PARALLELISM IN WRN

A major advantage of the relaxation-based methods is that they generate abundant parallelism (when the circuit is large and loosely coupled) and can be mapped to many

different types of parallel architectures. On the other hand, if the circuit is relatively small, as is the case for many of the examples used in this paper, a variety of techniques must be used to obtain large speedups. One straightforward way to parallelize WRN is to exploit the parallelism available at the subcircuit level, as is done in most other relaxation-based programs (for a review of these programs, see [20]). The subcircuits can be processed using either a Gauss-Seidel approach or a Gauss-Jacobi approach [24], as illustrated in Fig. 1. In the Gauss-Jacobi approach, all the subcircuits can be processed in parallel in each iteration because the computation associated with each subcircuit depends only on information generated in the previous iteration. However, the convergence speed is often much slower than using the Gauss-Seidel method, which uses the latest information wherever possible. Although the Gauss-Seidel algorithm is sequential in nature, there are opportunities for parallelism, as described below.

The processing sequence in our program uses a data-flow-based approach. However, a simplistic view is presented first to illustrate the parallelism that is inherent in the Gauss-Seidel method. Consider the circuit graph shown in Fig. 1(a). The nodes in the graph represent subcircuits, S_i , and the arcs represent connections between the subcircuits. In Fig. 1(b), the directed edges between the nodes imply a partial ordering or precedence relation between tasks. Therefore, $S_i \rightarrow S_j$ requires that S_i be completed before S_j is started. In fact, all predecessors must be completed before a successor begins execution. Subcircuits without any predecessors, except external inputs, are called initial tasks and they are always computed first. The width of the graph is the maximum size of any *independent* subset of tasks and it is this aspect that provides the parallelism in the Gauss-Seidel algorithm. Suppose we are at the k th iteration. It is obvious that S_1 , S_2 , and S_3 can be computed in parallel. After they finish, S_4 and S_5 can be computed in parallel. After they finish, not only can S_6 , S_7 , and S_8 be computed in parallel for their k th iteration, but also S_1 , S_2 and S_3 can be recomputed for their $(k + 1)$ th iteration. This is sometimes referred to as overlapping iterations or unrolling the task graph [20]. Therefore, a total of six processors can be kept busy. In a data-flow-based implementation, new tasks are generated whenever all dependences for a given task have been satisfied and this provides even more parallelism. However, the drawbacks of the GS approach are that the amount of parallelism available is limited and it is problem dependent.

A. Implementation of Parallelism at the Subcircuit Level

We have implemented a number of parallel versions of WRN in the PSPLAX program, which are described in this section. Task scheduling in PSPLAX is performed using a simple parallel task management system. All executable parallel tasks are kept in a task pool, and whenever a processor is ready to perform another task, it sim-

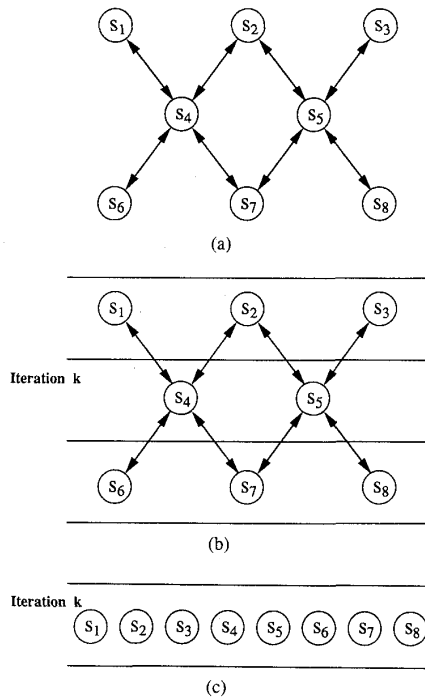


Fig. 1. Subcircuit-level parallelism (a) Original subcircuit task graph. (b) Gauss-Seidel ordering. (c) Gauss-Jacobi ordering.

ply obtains the next available task from the pool. When new tasks are generated, they are immediately placed in the pool so that idle processors can seize them and process them as soon as they are available. In PSPLAX, the task pool is implemented as a central task queue using a linked-list structure. Each entry of the queue contains a task function, task data, and a schedule time. Whenever a processor is free, it obtains exclusive access to the queue, removes a task from the queue, relinquishes the queue to other processors, and then processes the given task. When new tasks are generated, the processor will again try to gain exclusive control of the queue, add the new tasks to the queue, and then release the queue.

In PSPLAX, some tasks are added to the tail of the queue while others are added to the head of the queue to ensure that they get the highest possible priority for processing. For example, if a task is decomposed into a tree of subtasks, the leaves of the tree at level n are usually assigned the highest priority so that a task at level $n - 1$ can be started as soon as its leaves are completed, thereby minimizing the waiting time for all tasks [10]. One way to guarantee this is to assign each level a different priority and then, whenever there is a choice, process higher priority tasks before considering any lower priority task. In general, a set of queues with different priorities could be used to implement this feature and this leads to less contention when there are many processors. In our case, there are only two priorities and a few processors; therefore one queue with the ability to add tasks at the tail or head provides the desired characteristics with little or no

impact on the program code. Further details concerning the actual tasks and their priorities are provided in subsection III-B.

As described earlier, a task in the parallel implementation is simply the evaluation of a subcircuit for one relaxation iteration. The partitioner of SPLAX is used without modification in PSPLAX so that a meaningful comparison between the two programs can be performed. In addition, PSPLAX uses the Gauss-Seidel method since an earlier study [26] showed that, while Gauss-Jacobi scheme generates more parallelism, the overall convergence speed of WRN degrades significantly. Indeed, the convergence speed of Gauss-Seidel is superior to Gauss-Jacobi and it offsets the extra parallelism generated by the Gauss-Jacobi scheme on the example circuits we simulated. Of course, given enough processors, Gauss-Jacobi should, in principle, outperform Gauss-Seidel [17]. On the other hand, the study also showed that the parallel speedup with the Gauss-Seidel scheme is problem dependent. We concluded that for WRN it is very important to take advantage of the fast convergence speed of Gauss-Seidel and to improve the efficiency by exploiting other forms of parallelism.

The first version of PSPLAX exploits only subcircuit-level parallelism (SLP), and is named PSPLAX1.1. This algorithm, of course, is not new but is provided as a basis for comparison for the other parallel algorithms described later in the paper. The key portions of the program are provided below in pseudocode.

Algorithm 2: Parallel GS-WRN with SLP

```

dispatch( )
{
  repeat { /* initial tasks have already been scheduled
in Queue */
    while (Queue empty) wait;
    lock(Queue);
    take next task from Queue;
    unlock(Queue);
    execute task_function(task_data,task_time);
  } until ((Queue empty) AND (All processors idle))
}
wavesim(subcircuit,tstart) /* task_function */
{
  tstop ← setup_window(subcircuit,tstart);
  /* tstart = window start time */
  /* tstop = window end time */
  while (t ≤ tstop)
  {
    t ← pickstep(subcircuit, t);
    /* use GS-WN method */
    matrix_load(subcircuit, t);
    matrix_evaluate(subcircuit, t);
    matrix_ludecomp(subcircuit, t);
    RHS_load(subcircuit, t);
    matrix_fbsub(subcircuit, t);
  }
}

```

TABLE I
CIRCUIT STATISTICS FOR SIMULATION BENCHMARKS

Circuit Name	Nodes	Transistors	Subcircuits	Average Width	Graph Height	Feedback Loops
DECPLA	66	116	17	1.7	10	1
DECPLA.8	458	928	136	13.6	10	8
CRAMB	149	277	76	3.4	22	0
SCDAC	155	416	29	5.8	5	2
CKT3	312	428	70	4.1	17	1

```

lock(Queue);
foreach (fanoutsubcircuit of subcircuit)
  if (dependences of fanoutsubcircuit satisfied)
    schedule(fanoutsubcircuit, wavesim, tstart);
unlock(Queue);
}

```

The *dispatch()* routine is used to access the global queue, which contains all the tasks ready to be processed at any given time. Hence, every processor executes the *dispatch()* routine to obtain the next task. If there are no tasks to be performed (i.e., the queue is empty), the processor simply waits. Otherwise, it tries to gain exclusive control of the queue by calling the *lock()* routine. When it succeeds in locking the queue, it takes the next available task from the queue, relinquishes the queue using *unlock()*, and then processes the task. In PSPLAX1.1, there is only one task function, namely, *wavesim()*. This task function performs one waveform-Newton iteration on a specified subcircuit, in a given window, and then places the fan-in and fan-out subcircuits that are ready to be processed on the queue (those that have their input dependences satisfied). These are referred to as *fanoutsubcircuits* in the above algorithm. The *schedule()* routine takes as arguments the subcircuit, the function to be called, and the schedule time.

B. Experimental Results

Although no standard set of benchmark circuits has been defined for circuit simulation, our test suite consists of a number of industrial circuits that we have accumulated over the years. Table I provides a compiled set of statistics about the circuits. These include circuit size information, the number of subcircuits after circuit partitioning, the average width and height of the task graph after ordering, and the number of global feedback loops. Except for CRAMB, the circuits have one or two global feedback loops, although local feedback caused by floating capacitances exists between all neighboring nodes in all circuits. Additional information about the circuits is presented below.

Table II shows the results of PSPLAX1.1 running on various benchmark circuits of different sizes and characteristics. There are many measures of the performance of a parallel algorithm. The most commonly used indicator is the speedup of the parallel program (PSPLAX) with p

TABLE II
PARALLEL GS-WRN USING SLP

Circuit	Program Name	No. Proc Used	CPU Time	Actual Speedup	Relative Speedup
DECPLA	SPLAX PSPLAX	1	64.5s	1.0	1.1
		1	72.4s	0.9	1.0
		2	41.3s	1.6	1.8
		4	34.2s	1.9	2.1
DECPLA.8	SPLAX PSPLAX	1	521.0s	1.0	1.1
		1	580.9s	0.9	1.0
		2	292.5s	1.8	2.0
		4	150.7s	3.5	3.9
CRAMB	SPLAX PSPLAX	1	82.8s	6.3	7.0
		1	469.6s	1.0	1.1
		1	497.7s	0.9	1.0
		2	275.1s	1.7	1.8
CKT3	SPLAX PSPLAX	4	187.3s	2.5	2.7
		8	164.0s	2.9	3.0
		1	1022.4s	1.0	1.02
		1	1043.2s	0.98	1.0
SCDAC	SPLAX PSPLAX	2	542.4s	1.9	1.9
		4	316.0s	3.2	3.3
		8	238.7s	4.3	4.4
		1	496.2s	1.0	1.1
SCDAC	SPLAX PSPLAX	1	563.5s	0.9	1.0
		2	288.4s	1.7	2.0
		4	151.7s	3.3	3.7
		8	101.1s	4.9	5.6

processors over the sequential code (SPLAX) on one processor. For the rest of the paper, this factor is named actual speedup. Another important indicator is the speedup of PSPLAX with p processors over PSPLAX with one processor. This is a good indicator of how much parallelism is generated within a parallel algorithm and is called the relative speedup. A close examination of the actual speedup and the relative speedup will also yield information about the overhead of the parallel algorithm.

Table II contains results for the five circuits, as described in the following. These circuits are relatively small and therefore present some interesting problems in obtaining good speed improvements. DECPLA is a relatively small size circuit with over 100 MOS transistors. On one processor, PSPLAX1.1 achieves an actual speedup of 0.9. This number indicates that the overhead for the parallel implementation is about 10% of the total run time. For DECPLA, an actual speedup of 1.9 is achieved when the number of processors is 4 or greater. The maximum relative speedup is 2.1, indicating that, on

average, only two processors are busy during the simulation. The actual and relative speedups then level off. In fact, they actually decrease slightly. The reason is that PSPLAX1.1 can only utilize a maximum of four processors at any given time for this particular circuit. Any additional processors will only increase the likelihood of contention for the queue since there is no work for them to perform. Hence, algorithms that generate further parallelism would be useful for this circuit.

To estimate the maximum speedup that can be obtained under ideal or near ideal circumstances using PSPLAX1.1, we created the DECPLA.8 circuit, which contains exactly eight identical copies of DECPLA. The intention was to reduce the effects of improper load balancing on performance. The experiments performed on DECPLA.8 show that an actual speedup of 6.3 is achieved on DECPLA.8 with eight processors. The maximum relative speedup is close to 7.0. Of course, the reason the maximum relative speedup is not 8 derives from queue contention, locking, any residual improper load balancing, and other sources of parallel overhead.

The CRAMB circuit (about 500 MOS transistors) is much larger than DECPLA but is purely unidirectional circuit; that is, there are no global feedback loops. With the larger size comes broader path width, and this leads to higher efficiencies and better parallel speedups. The individual subcircuits in CRAMB are about the same size as the ones in DECPLA but there are more of them since the size of CRAMB is much larger. As a result, the speedups are higher than DECPLA. However, the relative speedup with eight processors is still low; moreover, we also observed a leveling off of the speedup when the number of processors reached 6. In short, there is still room for improvement by introducing more parallelism.

The next circuit is CKT3, which has 312 nodes and 428 transistors. On eight processors, we obtained an actual speedup of 4.3 and a relative speedup of 4.4. In this case, the task granularity is much larger; i.e., the subcircuits are larger. Hence, further speedup is also expected here with additional forms of parallelism. SCDAC is roughly the same size as CRAMB except that the path width of SCDAC is much broader than CRAMB. Hence, a better performance was expected with PSPLAX1.1 and, indeed, with eight processors, PSPLAX1.1 achieves an actual speedup of 4.9 and a relative speedup of 5.6. These values approach the maximum speedups set by DECPLA.8. In view of this, further speedup is not likely with the addition of other forms of parallelism.

To summarize, the results in Table II show that PSPLAX1.1 performs well on large circuits with large task graph widths and poorly on the circuits that have narrow task graphs or large variations in the subcircuit sizes. Clearly, when there are fewer subcircuits than processors, the efficiency is low since a certain number of processors will always be idle during the entire simulation. For circuits with differing subcircuit sizes, the main problem is load balancing. If, in a given circuit, S_1 and S_2 are in the same rank and S_1 is much larger than S_2 , then the proces-

or assigned to S_2 will finish processing it much earlier than the processor working on S_1 . As a result, the second processor, as well as other processors, may be waiting for a long time for the first processor to finish S_1 before new tasks are placed on the queue. To achieve better efficiency on a parallel computer, it is desirable, if possible, to have equal sized tasks, but often this is not possible because of other considerations (convergence, in our case). The other option is to decompose large tasks into many smaller tasks so that more tasks are made available for any free processors. We use the latter approach to improve performance in our program.

III. IMPROVING THE PERFORMANCE OF PARALLEL GS-WRN

In this section, two other forms of parallelism are added to improve the performance of PSPLAX: the parallel time point (PTP) method [25]–[27] and parallel model evaluation (PME). Then, a few techniques are introduced to fine-tune the performance. Each approach is tested on the same benchmark circuits, and the results are analyzed with respect to the tuning parameters.

A. Parallel Time Points

The waveform-relaxation-Newton method requires, for each subcircuit, forming and solving a matrix equation at each time point within a given window. Owing to the time point selection strategy, most of the computations at these mesh points can be performed in parallel. For example, assume that there are four time points in some interval $[0, T]$, as shown in Fig. 2. The computation at each point involves evaluating and loading the matrix and right-hand-side (RHS) terms for the matrix equation, followed by an LU decomposition step and a forward/back substitution operation. In order to perform these tasks in parallel across time points, the position of the time points and the values of the required waveforms must be known in advance. The step refinement strategy for waveform-relaxation-Newton described earlier implies that the number of time points, m , and their location are known before beginning the computation of the waveform for v_i at the $(k + 1)$ st iteration. Also, the set of waveforms necessary to compute the $(k + 1)$ st iteration are known in advance. Therefore, for a given interval, the matrices and portions of the RHS vector for a subcircuit can be computed at all m time points in parallel. In addition, the LU decomposition operation for each matrix can be performed in parallel across the time points.

The operations that remain at each time point are to load the portion of the RHS that depends on the solution at the previous time point and to perform the forward-elimination and back-substitution steps. These tasks cannot be done in parallel across time points. Since the tasks at each time point consist of a parallel portion and a sequential portion, it is convenient to divide them into two smaller tasks. The two parts are as follows:

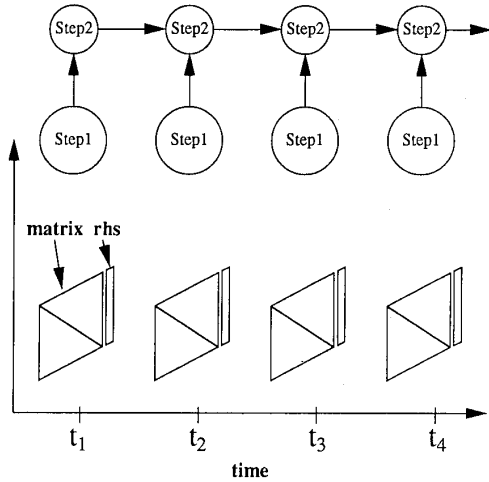


Fig. 2. Parallel time point approach.

- Step 1: Matrix evaluation, and partial RHS evaluation, *LU*-decomposition (parallelism across time points).
- Step 2: Remaining RHS evaluation, forward and back substitution (sequential across time points).

The task dependences are illustrated in Fig. 2. Note that the step 2 task at a point t_n must be completed before the starting step 2 task at t_{n+1} . Hence, this portion is done sequentially across time points. However, this represents only a small percentage of the total cost of each solution relative to the other operations. For example, in the CRAMB circuit of Table I, the loading of the remaining portion of the RHS is a simple *for*-loop involving only a few multiplications and additions for each entry and requires less than 1% of the total time. The forward and backward substitution consumes only 4.5% of the cost of each Newton iteration when solving the entire 150×150 matrix. Therefore, for the small submatrices encountered after partitioning such a circuit, the required time for step 2 is well below 5%. Since this approach effectively allows all the time points in a window for a given subcircuit to be processed in parallel, it is called the parallel time point (PTP) algorithm.

1) *Implementation of the Parallel Time Point Algorithm*: As in many other parallel algorithms, PTP trades off memory usage and task granularity for increased parallelism. In PSPLAX1.1, each task in the queue involved solving for the waveforms for an entire window for one particular subcircuit. However, in order to add PTP, each time point within the window must be processed separately. Without PTP, each subcircuit required enough memory space to store only *one* matrix equation, and after the solution for a time point was obtained, the same matrix was reused for the next time point. However, with PTP, a number of matrices must be allocated for each window per subcircuit since the time points are processed in parallel. For example, if eight processors are working on the same subcircuit for iteration k , eight

different time points can be computed in parallel; therefore eight different matrices must be allocated and used by the processors. This results in a sharp increase in memory usage. However, not all the time points in a window need to be processed in parallel; PTP is required to generate only enough parallelism to keep all the processors busy. As an example, consider the circuit graph of Fig. 1. For this specific case, only four parallel time points are needed for a machine with eight processors since the minimum path width of the circuit is 2. In other words, at any two given time points, at least two subcircuits will be processed in parallel; therefore, processing four time points for each window is sufficient to keep all eight processors busy. While overly simplistic, this example illustrates one way to select the number of time points to compute in parallel. The actual selection is more complicated and is described in a later section.

The PTP algorithm has been implemented and the resulting code is named PSPLAX1.2. The critical portions of PSPLAX1.2 are shown in pseudocode form in Algorithm 3.

Algorithm 3: Parallel GS-WRN with SLP/PTP

```

dispatch( )
{
  repeat { /* initial tasks have already been schedule
in Queue */
    while (Queue empty) wait;
    lock(Queue);
    take next task from Queue;
    unlock(Queue);
    execute task_function(task_data,task_time);
  } until ((Queue empty) AND (All processors idle))
}

wavesim(subcircuit,tstart) /* task_function 1. */
{
  tstop ← setup_window(subcircuit,tstart);
  t0 ← tstart;
  for (n = 1 to MATNUM) { /* schedule up to MAT-
NUM time points */
    tn ← pickstep(tn-1); /* never goes beyond tstop
*/
    lock(Queue);
    schedule(subcircuit, step1, tn);
    unlock(Queue);
    if(tn ≥ tstop) return; /* scheduled whole window
*/
  }
}

step1(subcircuit, tn) /* load matrix and part of RHS */
{
  matrix_load(subcircuit, tn);
  matrix_evaluate(subcircuit, tn);
  partial_RHS_load1(subcircuit, tn);
  if ((step2 at tn-1 completed) AND (step2 at tn not
scheduled)) {

```

```

lock(Queue);
schedule(subcircuit, step2, tn);
unlock(Queue);
}
}
step2(subcircuit, tn) /* fill rest of RHS and do fbsub */
{
partial_RHS_load2(subcircuit, tn);
matrix_fbsub(subcircuit, tn);
if ((step1 at tn+1 completed) AND (step2 at tn+1 not
scheduled)) {
lock(Queue);
schedule(subcircuit, step2, tn+1);
unlock(Queue);
}
}
if (tn ≥ tstop) { /* finished window */
lock(Queue);
foreach (fanoutsubcircuit of subcircuit)
schedule(fanoutsubcircuit, wavesim, tstart);
unlock(Queue);
}
else { /* need to add at least one more point in this
window */
/* pick next timepoint, after last one scheduled
so far */
/* use MATNUM to find the last point */
tn+MATNUM ← pickstep(subcircuit, tn+MATNUM-1);
lock(Queue);
schedule(subcircuit, step1, tn+MATNUM); /* add
only one more at the end */
unlock(Queue);
}
}
}

```

Like PSPLAX1.1, PSPLAX1.2 also uses a global management routine, *dispatch()*, which performs the same function as in PSPLAX1.1. However, the task of simulating the individual subcircuit is now separated into *step1()* and *step2()*, whose functions were described above. The *wavesim()* routine now serves as a function generator for the time points to be simulated. It schedules all the *step1()* functions for a subcircuit in a window. When *step1()* is completed, *step2()* is scheduled when all the necessary information is available from the previous time point (see Fig. 2). The maximum number of time points actually solved in parallel is given by *MATNUM*.

2) *Experimental Results*: PSPLAX1.2 was used to simulate the same benchmark circuits as PSPLAX1.1 and the results are shown in Table III. For DECPLA, the first thing to notice is that the actual speedup with one processor goes down from 0.9 of PSPLAX1.1 to 0.8. This is expected since the addition of PTP decreases the granularity of the tasks and increases the overhead. However, with eight processors, an actual speedup of 2.2 is achieved, as opposed to 1.9 with PSPLAX1.1. Hence, the addition of PTP produces an improvement in this case.

TABLE III
PARALLEL GS-WRN WITH SLP/PTP

Circuit	Program Name	No. Proc Used	CPU Time	Actual Speedup	Relative Speedup
DECPLA	SPLAX	1	64.5s	1.0	1.3
		1	82.6s	0.8	1.0
	PSPLAX	2	46.7s	1.4	1.8
		4	32.5s	2.0	2.5
DECPLA.8	SPLAX	1	521.0s	1.0	1.3
		1	667.8s	0.8	1.0
	PSPLAX	2	367.6s	1.4	1.8
		4	191.8s	2.7	3.5
CRAMB	SPLAX	1	469.6s	1.00	1.22
		1	573.3s	0.8	1.0
	PSPLAX	2	314.3s	1.5	1.8
		4	177.1s	2.7	3.2
CKT3	SPLAX	1	1022.4s	1.0	1.2
		1	1241.0s	0.8	1.0
	PSPLAX	2	642.9s	1.6	1.9
		4	357.1s	2.9	3.5
SCDAC	SPLAX	1	496.2s	1.0	1.3
		1	662.0s	0.8	1.0
	PSPLAX	2	336.6s	1.5	2.0
		4	177.8s	2.8	3.7
	SPLAX	1	107.0s	4.6	6.2
		1	107.0s	4.6	6.2

However, the increase in speedup is still relatively small. This may be due in part to the sequential element, *step2()*, but mainly because there are too few time points in most of the window iterations (especially during the first few iterations) to make effective use of the PTP method.¹ However, the point that should not be overlooked is that the speedups are not leveling off, as they did in the SLP case.

The results of running DECPLA.8 indicate that, with eight processors, the maximum actual speedup is about 5.1 and the maximum relative speedup is about 6.5. An improvement in the actual speedup is not expected since DECPLA.8 generates enough work for PSPLAX1.1 with eight processors, and the additional PTP algorithm can only introduce more overhead, thus reducing the performance. This suggests that, for circuits with enough subcircuit level parallelism, PTP is not necessary and should not be used. Although the width of the task graph provides some insight into making this decision automatically, load-balancing issues and the number of time points used during the computation make it difficult to do this optimally.

With parallel time points, the performance on CRAMB is also improved and the actual speedup increases from 2.9 to 3.4 with eight processors. However, the relative speedup is 4.1, which shows that there is still room for improvement. Note that the actual speedup values on CRAMB do not level off as sharply as in the case of

¹For a detailed analysis, see the section below that discusses the effect of varying *MATNUM*.

PSPLAX1.1, indicating that a further speedup is possible with PSPLAX1.2 with the addition of more processors. For CKT3, PSPLAX1.2 shows a slight improvement in actual speedup with eight processors over PSPLAX1.1. Again, we observe a better relative speedup and a more gradual leveling off of the actual speedup, indicating that additional processors can be used effectively.

PSPLAX1.2 exhibits a degradation in the actual speedup on SCDAC with eight processors compared with PSPLAX1.1. This result is again expected since PSPLAX1.1 already gives an actual speedup of 4.9 and a relative speedup of 5.6. The additional parallel tasks of PTP are overwhelmed by the additional overhead. This reasoning is confirmed by a better relative speedup of 6.2 with PSPLAX1.2. In summary, we observed improvements in all but one case with the addition of PTP, but perhaps most importantly, the speedup did not level off in any of the cases (up to eight processors), indicating a potential for better performance with more processors.

3) *The Effect of Window Size:* With the PTP algorithm, the size of each window has a greater impact on the performance than it does when only subcircuit-level parallelism is used. Recall that the time point selection strategy places very few time points in each window during the first few iterations. In fact, during the first iteration, there is only one time point in the entire window; hence, there are no parallel time points to compute during the first iteration. Only during the subsequent iterations, as more and more time points are added, does the PTP algorithm make significant contributions toward improving the performance. If the window size is small, then the likelihood is higher that the waveforms for the present window will converge before a significant number of time points are added to the window. Hence, to take full advantage of the PTP algorithm, the window size has to be relatively large compared with the one used in the SPLAX and PSPLAX1.1, yet small enough not to cause significant convergence degradation.

The effect of window size (WS) on performance is illustrated in the experimental results plotted in Figs. 3 and 4 for DECPLA and SCDAC, respectively. To properly assess the effect of window size, PSPLAX1.1 was also executed with the same window size parameters. The windows, given in increasing sizes, are $WS1$, $WS2$, $WS3$, and $WS4$, and in each case the original window size ($WS1$) has been multiplied by a factor of 1, 2, 3 and 4, respectively. As seen in Figs. 3 and 4, the performance of PSPLAX1.1 shows a general trend of slowing down when the window size increases. With PSPLAX1.2, a much greater improvement is observed from $WS1$ to $WS2$ because the performance of PSPLAX1.2 depends on the number of time points that can be processed in parallel. Unfortunately, as the window size increases further, to $WS3$ and $WS4$, the benefit of exploiting multirate behavior and the parallel time point computation is offset by the slower convergence speed. These results demonstrate that the optimal window size for the PTP approach is larger than the optimal window size for SPLAX or PSPLAX1.1.

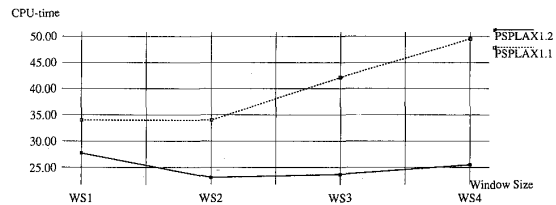


Fig. 3. Effect of window size on performance of DECPLA.

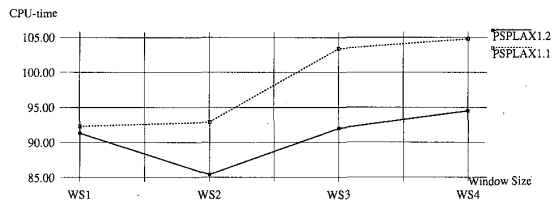


Fig. 4. Effect of window size on performance of SCDAC.

4) *Choosing the Number of Parallel Time Points:* The parallel time point algorithm provides another degree of freedom. The number of time points that are actually computed in parallel can be adjusted. The problem with allowing too many parallel time points is that each computation requires a separate matrix; therefore, the memory requirements for this approach quickly overwhelm the system. In the next set of experiments, plotted in Figs. 5 and 6 for DECPLA and SCDAC, respectively, the number of matrices available to each subcircuit ($MATNUM$) is varied from 1 to 4 to observe how the number of allowed parallel time points affects the performance of PSPLAX1.2.

The results obtained by simulating DECPLA in Fig. 5 show two effects. The first is that if $MATNUM > 1$, enough parallelism is generated by the PTP algorithm to improve the performance. The second is that if $MATNUM$ is too large, the number of accesses to the queue increases and this begins to reduce the overall performance because of queue contention and additional overhead in queue management. The same experiments repeated with SCDAC, shown in Fig. 6, confirm these statements. Therefore, $MATNUM$ should be large enough to generate the necessary amount of parallelism for a given parallel computer when simulating a particular circuit, but small enough so that unnecessary overhead and memory requirements are not introduced. In view of this, we set $MATNUM$ to a value based on the number of processors available and the minimum task graph width, as follows:

$$MATNUM = \frac{\text{no. of processors}}{\text{min. task graph width}}$$

B. Addition of Parallel Model Evaluation

A well-known source of fine-grain parallelism is parallel model evaluation (PME), and this has been used in almost every existing parallel circuit simulation code. At each time point, a subcircuit computation involves a ma-

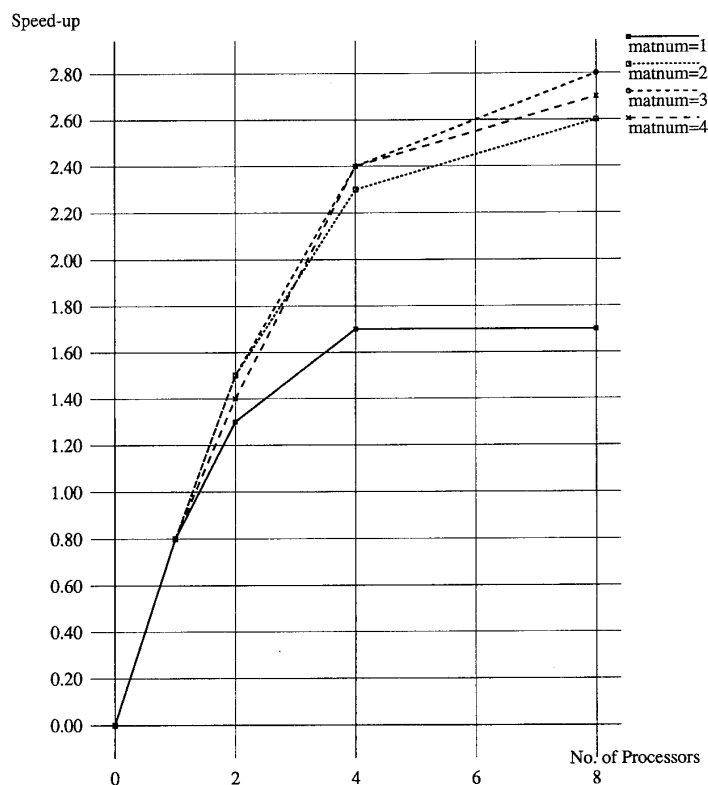


Fig. 5. Effect of MATNUM on DECPLA.

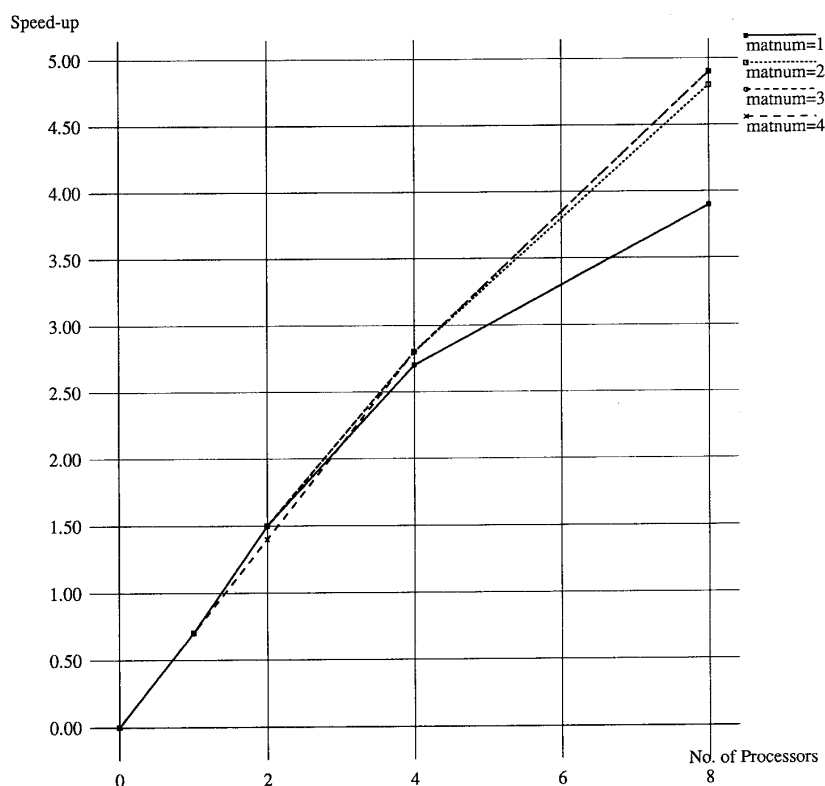


Fig. 6. Effect of MATNUM on SCDAC.

TABLE IV
PARALLEL GS-WRN WITH SLP/PTP/PME

Circuit	Program Name	No. Proc Used	CPU Time	Actual Speedup	Relative Speedup
DECPLA	SPLAX	1	64.5s	1.0	1.4
	PSPLAX	1	88.0s	0.7	1.0
		2	49.1s	1.3	1.8
		4	31.6s	2.0	2.8
		8	27.7s	2.3	3.2
DECPLA.8	SPLAX	1	521.0s	1.0	1.3
	PSPLAX	1	697.5s	0.8	1.0
		2	381.4s	1.4	1.8
		4	201.3s	2.6	3.5
		8	115.9s	4.5	6.0
CRAMB	SPLAX	1	469.6s	1.0	1.4
	PSPLAX	1	650.6s	0.7	1.0
		2	343.0s	1.4	1.9
		4	212.9s	2.2	3.1
		8	162.5s	2.9	4.0
CKT3	SPLAX	1	1022.4s	1.0	1.3
	PSPLAX	1	1331.1s	0.8	1.0
		2	693.6s	1.5	1.9
		4	375.6s	2.7	3.5
		8	249.0s	4.1	5.4
SCDAC	SPLAX	1	496.2s	1.0	1.4
	PSPLAX	1	709.6s	0.7	1.0
		2	361.1s	1.4	2.0
		4	191.4s	2.6	3.7
		8	117.3s	4.2	6.1

trix load and a linear equation solution. The use of PME effectively forms the matrix equation in parallel. In our case, we have many time points being computed simultaneously, so the amount of parallelism that can be generated using this approach is $MATNUM \times (\text{no. of devices})$. Of course, the speedup observed at each time point is limited by the percentage of time required for model evaluation compared with that for linear equation solution. The linear equation can also be solved in parallel, but this form of fine-grain parallelism is not currently implemented in PSPLAX.

The task of forming the matrix and RHS can be divided into many smaller tasks, each of which consists of computing the contribution of several devices to the appropriate entries of the matrix equation. These tasks are subsequently placed at the head of the queue to give them higher priority than subcircuit tasks or time point tasks. The idea here is to compute the model tasks as quickly as possible so that step 2 can be started as soon as possible since it is part of the sequential bottleneck. Note that there is a sequential element in PME. As the contributions are computed, they have to be added to each element of the matrix. This accumulation is inherently a serial process. There are a variety of ways of performing this operation efficiently [6], [18]. In our case, a single lock per row of the matrix is used to guarantee that only one processor is updating a particular element of the matrix at a time. The resulting program is named PSPLAX1.3. The results of running PSPLAX1.3 is shown in Table IV.

The addition of the parallel model evaluation adds ad-

ditional overhead to the implementation. Only in DECPLA is PSPLAX1.3 able to show marginal improvement over PSPLAX1.2. The problem is that the granularity of the computation is small relative to the cost of locking and queue access, and the large number of tasks generated by the approach aggravates the queue contention problem further. To avoid generating excessive amounts of parallelism, the option of parallel model evaluation is activated during the simulation only when there are idle processors. When a processor is ready to evaluate the entries of a matrix equation, it first checks to see if there are any processors available. If there are, the processor places the PME tasks on the queue; otherwise, the processor simply evaluates and loads the entries of the matrix equation itself. With this modification, the speedups are always greater than or equal to the values given in Table III. Of course, if the overhead of locking and queue access could be reduced, PME could always be invoked; however, given the characteristics of our implementation, it should only be used when absolutely necessary. On larger machines with more processors, better speedups are expected using PSPLAX1.3.

IV. CONCLUSIONS

A number of approaches of differing granularity have been described and implemented for the parallelization of waveform-relaxation-Newton on the Alliant multiprocessor. The experimental results presented here show that PSPLAX1.1, the version of the GS-WRN algorithm which exploits only subcircuit-level parallelism, has low overhead but does not produce good speedups if the task graph is not wide enough to keep all processors busy. A significant improvement is made with PSPLAX1.2, which exploits parallelism at the subcircuit level and also uses the parallel time point approach. This version is able to generate enough parallelism when the task graph is narrow, but the performance relies on an appropriate choice of window size. In fact, larger windows are necessary to obtain the best performance. The addition of parallel model evaluation did not significantly improve the performance because of the excessive overhead in the current implementation. If the overhead could be reduced, this approach would be more effective because it generates up to $(\text{no. of time points}) \times (\text{no. of devices})$ parallel tasks. This permits better load balancing and reduces the starvation conditions that are often encountered in the SLP code.

It is important to emphasize that SLP, PTP, and PME are not in competition with each other and that all of them can, and should, be exploited simultaneously to speed up the simulation process if enough processors are available. SLP should be used on ranks in the task graph that already provide enough parallelism to satisfy the demands of the system. PTP should be invoked in the ranks where there is not enough inherent parallelism. PME should be used on any large subcircuits or whenever any processors are idling. In summary, the best approach for parallel WRN is to combine SLP, PTP, and PME, as is done in

PSPLAX1.3, and then invoke each method as needed, given a finite number of processors and a specific task graph. We are currently adjusting the aforementioned tuning parameters to optimize performance and are developing heuristics to select these parameters automatically.

The results presented in this paper demonstrate clearly that the performance should continue to improve as more processors are added, since the speedups are not leveling off at eight processors in any of the example circuits. In fact, the limiting factor in the speedup will probably be the maximum number of processors that can be added to a shared memory machine, such as the Alliant, not any inherent limitations in the PSPLAX program. Finally, other techniques such as time segment pipelining [16] and time point pipelining [15] could be added to PSPLAX in addition to the existing techniques to further enhance the performance.

ACKNOWLEDGMENT

The authors would like to thank K. Gallivan for the many useful discussions on this topic. The paper reviewers also provided excellent feedback to improve the quality of the manuscript, and thanks are due to them for their diligence. R. Newton, D. Webber, J. White, and A. Sangiovanni-Vincentelli also provided useful suggestions in the initial stages of this research.

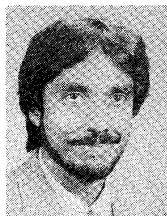
REFERENCES

- [1] L. W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Electronics Research Laboratory Rep. No. ERL-M520, University of California, Berkeley, May 1975.
- [2] A. R. Newton and A. Sangiovanni-Vincentelli, "Relaxation-based circuit simulation," *IEEE Trans. Computer-Aided Design*, vol. CAD-3, pp. 308-330, Oct. 1984.
- [3] J. White and A. S. Vincentelli, *Relaxation Techniques for the Simulation of VLSI Circuits*. Norwell, MA: Kluwer, 1986.
- [4] R. Saleh and A. R. Newton, "The exploitation of latency and multirate behavior using nonlinear relaxation for circuit simulation," *IEEE Trans. Computer-Aided Design*, pp. 1286-1298, Dec. 1989.
- [5] G. K. Jacob, A. R. Newton, and D. O. Pederson, "Direct-method circuit simulation using multiprocessors," in *Proc. IEEE Int. Symp. Circuits Syst.* (San Jose, CA), May 1986, pp. 170-173.
- [6] G. Bischoff and S. Greenberg, "CAYENNE: A parallel implementation of the circuit simulator SPICE," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1986, pp. 182-185.
- [7] P. Cox, R. Burch, D. Hocevar, and P. Yang, "SUPPLE: Simulator utilizing parallel processing and latency exploitation," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1987, pp. 368-371.
- [8] P. Sadayappan and V. Visvanathan, "Circuit simulation on a multiprocessor," in *Proc. Custom Integrated Circuit Conf.*, May 1987, pp. 124-128.
- [9] C-P. Yuan, R. Lucas, P. Chan, and R. Dutton, "Parallel electronic circuit simulation on the iPSC system," in *Proc. IEEE 1988 Custom Integrated Circuit Conf.*, May 1988, pp. 6.5.1-6.5.4.
- [10] M-C. Chang and I. N. Hajj, "iPRIDE: A parallel integrated circuit simulator using direct method," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1988, pp. 304-307.
- [11] O. Wing and J. W. Huang, "A computation model of parallel solution of linear equations," *IEEE Trans. Comput.*, vol. C-29, pp. 632-638, 1980.
- [12] G. K. Jacob, A. R. Newton, and D. O. Pederson, "Parallel linear-equation solution in direct-method circuit simulation," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 1987, pp. 1056-1059.
- [13] P. Sadayappan and V. Visvanathan, "Parallelization and performance evaluation of circuit simulation on a shared-memory multiprocessor," in *Proc. Int. Conf. Supercomputing*, May 1988, pp. 254-265.
- [14] D. Smart and J. White, "Reducing the parallel solution time of sparse circuit matrices using reordered Gaussian elimination and relaxation," in *Proc. Int. Symp. Circuits Syst.*, June 1988.
- [15] J. White, R. Saleh, A. Sangiovanni-Vincentelli and A. R. Newton, "Accelerating relaxation algorithms using waveform Newton, step refinement and parallel techniques," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1985, pp. 5-7.
- [16] D. Dumlugol, P. Odent, J. Cockx, and H. De Man, "The segmented waveform relaxation method for mixed-mode switch electrical simulation of digital MOS VLSI circuits and its hardware acceleration on parallel computers," in *Proc. Int. Conf. Computer-Aided Design*, Sept. 1986, pp. 84-87.
- [17] D. Smart and T. Trick, "Increasing parallelism in multiprocessor waveform relaxation," in *Proc. 1987 Int. Conf. Computer-Aided Design*, Nov. 1987, pp. 360-363.
- [18] K. A. Gallivan, P. Koss, S. Lo, and R. A. Saleh, "A comparison of parallel relaxation-based circuit simulation techniques," in *Proc. Electro/88*, May 1988.
- [19] J. T. Deutsch and A. R. Newton, "MSPLICE: A multiprocessor-based circuit simulator," in *Proc. Int. Conf. Parallel Processing*, June 1984, pp. 207-214.
- [20] R. Saleh *et al.*, "Parallel circuit simulation on supercomputers," *Proc. IEEE*, pp. 1915-1931, Dec. 1989.
- [21] R. Saleh and J. White, "Accelerating relaxation-based circuit simulation using waveform-Newton and step refinement," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 951-958, Sept. 1990.
- [22] Alliant Computer Systems Corp., *FX/Series—Architecture Manual*, 1986.
- [23] L. Kantorovich and G. Akilov, *Functional Analysis in Normed Spaces* (translated by D. Brown and A. Robertson). Oxford: Pergamon Press, 1964.
- [24] R. Varga, *Matrix Iterative Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1962.
- [25] R. Saleh, D. Webber, E. Xia, and A. Sangiovanni-Vincentelli, "Parallel waveform-Newton algorithms for circuit simulation," in *Proc. IEEE Int. Conf. Comp. Design*, Oct. 1987, pp. 660-663.
- [26] E. Xia, "Parallel waveform-Newton algorithms for circuit simulation," M.S. thesis, University of Illinois, Center for Supercomputing Research and Development, Apr. 1988.
- [27] P. Odent, L. Claesen, and H. De Man, "A combined waveform relaxation-waveform relaxation Newton algorithm for efficient parallel circuit simulation," in *Proc. Int. Symp. Circuits Syst.*, May 1990, pp. 244-248.



Eugene Z. Xia received the B.S. degree in electrical engineering and mathematics from the Virginia Polytechnic Institute and State University in 1986 and the M.S. degree in computer science from the University of Illinois at Champaign-Urbana in 1988.

He was with the Digital Equipment Corporation in Marlborough, MA, for three years and is now a Ph.D. student at the University of Maryland.



Resve A. Saleh (S'79-M'86) obtained the B.Eng. degree (electrical) from Carleton University, Ottawa, Canada, in 1979, and the M.S. and Ph.D. degrees from the University of California at Berkeley in 1983 and 1986, respectively.

He has worked in industry for a number of companies, including the Mitel Corporation, Tektronix, and the Toshiba Corporation. Since 1986, he has been a faculty member at the University of Illinois at Champaign-Urbana, where he is currently directing research in mixed-mode simulation, parallel processing, and analog CAD and synthesis.

Dr. Saleh has served on the technical committees of the Custom Integrated Circuits Conference and the Design Automation Conference since 1987 and was on the technical committee of the International Conference on Computer Design in 1991. He was the recipient of a 1990 Presidential Young Investigator Award and an Inventors Recognition Award from the Semiconductor Research Corporation for his work on the iSPLICE3 simulator. He is also the coauthor, with Prof. A. R. Newton, of the book *Mixed-Mode Simulation*, which was published in 1990.