NONLINEAR RELAXATION ALGORITHMS FOR CIRCUIT SIMULATION

by

Resve A. Saleh

Memorandum No. UCB/ERL M87/21

15 April 1987

COVER PAGE

224

NONLINEAR RELAXATION ALGORITHMS FOR CIRCUIT SIMULATION

by

Resve A. Saleh

Memorandum No. UCB/ERL M87/21

15 April 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

NONLINEAR RELAXATION ALGORITHMS FOR CIRCUIT SIMULATION

by

Resve A. Saleh

Memorandum No. UCB/ERL M87/21

15 April 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ABSTRACT

Circuit simulation is an important Computer-Aided Design (CAD) tool in the design of Integrated Circuits (IC). However, the standard techniques used in programs such as SPICE result in very long computer run times when applied to large problems. In order to reduce the overall run time, a number of new approaches to circuit simulation have been developed and are described in this dissertation. These methods are based on nonlinear relaxation techniques and exploit the relative inactivity of large circuits. Simple waveform processing techniques are described to determine the maximum possible speed improvement which can be obtained by exploiting this property of large circuits. Three simulation algorithms are described, two of which are based on the Iterated Timing Analysis (ITA) method and a third based on the Waveform-Relaxation Newton (WRN) method. New programs which incorporate these techniques have been developed and used to simulate a variety of industrial circuits. The results from these simulations are also provided. The techniques are shown to be much faster than the standard approach. In addition, a number of parallel aspects of these algorithms are described and a general space-time model of parallel task scheduling is developed.

ACKNOWLEDGEMENTS

I thank Prof. A. Richard Newton, my research advisor for both the Masters and Ph.D. degrees, for his encouragement and guidance during the course of this work. He provided me with many opportunities to work on a variety of research projects over the past few years, both at the University of California and at a number of industrial locations. His dedication and dynamic energy inspired me to do my best work and I thank him for one of the most enjoyable working relationships I've ever had. Based on my experiences in thesis writing, I present the following theorem, without proof, for his other students, both now and in the future: A Ph.D. thesis is guaranteed to converge in a finite number of "Newton"

iterations if, and only if, the initial draft is close enough to the final dissertation.

I also thank Prof. Alberto Sangiovanni-Vincentelli who provided me with a lot of assistance and motivation (both on and off the tennis court). His door was always open for discussions, and he contributed many useful ideas to this work. Prof. Don O. Pederson was a constant source of inspiration and offered me a lot of good advice during the course of my graduate studies. I thank him for giving me the benefit of his many years of experience and, along with Prof. Newton and Prof. Sangiovanni-Vincentelli, for providing an excellent research environment for the students in the Berekely CAD group. I also thank Prof. Ole Hald for reviewing this dissertation.

A number of graduate students in the CAD group made substantial contributions to the body of work presented in this dissertation. Jacob White, Ken Kundert, and Peter Moore were all involved in the development of the SPLICE3 program. I enjoyed many fruitful discussions and collaborations with Jacob White and I thank him for all his help and his insights into the theoretical aspects of the work. Those who provided other programming assistance, engaged in useful discussions and read early versions of this dissertation were Giorgio Casinovi, Ron Gyurscik, Tammy Huang, Seung Hwang, George Jacob, Young Kim, Ken Kundert, Tony Ma, Kartikeya Mayaram, Peter Moore, Tom Quarles, Rick Spickelmier, Don Webber, Nick Weiner and Jacob White.

Jeff Deutsch, George Jacob and Morgan Hua helped in the development of the parallel version of SPLICE3. Jeff Deutsch also spent many long hours discussing parallel aspects of this work with me and I thank him for his contribution. A number of people at Shiva Multisystems also deserve mention for help on parallel processing aspects. In particular, Bob Floyd, Dierdre Ryan, Howard Ko, John Chan and Dileep Devekar provided software support and technical assistance. In addition, Susan Eggers at U.C. Berkeley and Dave Smart and Kyle Gallivan of the University of Illinois contributed to the chapter on parallel processing.

Tai Sato, Takayasu Sakurai, Shuji Ohtsubo, Nobu Matsumoto, Kiichiro Tamaru, Yukimasa Uchida, Tetsuya Iizuka, and Takeshi Shima of Toshiba Corporation provided technical support and an excellent working environment for the development of the SPLICE2 program and Hiroyuki Kinoshita provided the modified routines for the MOS level 3 model now used in SPLICE2 and SPLICE3, and a number of other programs at U.C. Berkeley. To these and other friends in Japan, "Domo arigato gozaimashita!". I would also like to thank K. Shimizu and Toshiba Corporation for the financial support and making it possible to spend an exciting and memorable summer in Japan. Jim Kleckner, of SDA systems, provided me with a lot of help in the early stages of this project with programming and useful discussions about the SPLICE2 program and I thank him for his assistance and encouragement.

Finally, I thank my wife, Lynn, my parents, Ehsanes and Shahid-ara Saleh, and John and Rosemary Hilchie for all their moral support during the course of my graduate studies and beyond.

Funding for this research was provided by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the Hewlett-Packard Company and Toshiba Corporation, and computer resources were provided by Digital Equipment Corporation.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Circuit simulation continues to be an important tool in the design of Integrated Circuits (IC). It is certainly one of the most heavily used Computer-Aided Design (CAD) tool in terms of CPU-time in the IC design cycle. The success of this form of simulation is primarly due to its reliability and its ability to provide precise electrical waveform information for circuits containing complex devices and all associated parasitics. This detailed form of analysis with guaranteed accuracy is usually referred to as *circuit-level simulation*. A number of higher-level simulation tools have also been used [New78a, Bry80, Hil80, Sak81, Kle84, Rao85] to verify circuit functionality and to obtain first-order timing characteristics. These techniques were developed to cope with the ever-increasing number of devices integrated on a single silicon chip. While these higher-level tools provide enough information to design working circuits, there is still a significant time lag between a functioning circuit and a circuit which meets the design specifications, particularly in the case of high-performance custom integrated circuits. In fact, circuit simulation is the only tool which provides enough detail to ensure that circuits of this type will meet specifications over a wide range of operating conditions.

At the present time, the most popular circuit simulation tool is the SPICE2 program [Nag75] developed in the early 1970's at the University of California at Berkeley. There are many thousands of copies of this program in use, as well as a number of versions of "alphabet-SPICE" (e.g., HSPICE, PSPICE, GSPICE) being marketed commercially. This program offers a wide variety of analysis types including DC solution, time-domain transient analysis, AC analysis, noise and distortion. Of these, the time-domain transient analysis is the most computationally expensive in terms of CPU-time.

The SPICE program was originally designed to simulate circuits containing up to 100 transistors. However, at some companies it has been routinely used to simulate circuits containing over 10,000 transistors at great expense! The program is accessed over 50,000 times per month at a number of companies with a "job mix" that conforms to the 80-20 rule. That is, 80% of the SPICE runs are small circuits which consume only 20% of the total CPU-time used each month, while 20% of the jobs are very large and consume 80% of the CPU-time used each month.

The circuit simulation problem in the time-domain involves the solution of a system of nonlinear first-order ordinary differential equations. The standard approach to circuit simulation uses *direct methods* to solve the circuit equations. Briefly, a numerical integration method is used to to convert the nonlinear differential equations into a set of nonlinear difference equations, which are then converted to linear equations using the Newton method, and solved using a sparse LU decomposition technique. There are two limitations in this approach which make it somewhat inappropriate for large circuits. A fundamental problem is that the sparse linear solution dominates the run time for large circuits [NeSa83]. The other limitation is due to the fact that, at each time point, all variables in the system are solved using a common time-step based on the fastest changing component in the system, even though some components may be changing very slowly [NeSa83]. This can be inefficient for both small and large circuits, but it is more significant for very large problems where most of the components are either changing very slowly or not changing at all.

A variety of techniques have been investigated to improve the performance of circuit simulators. Current research can be broadly classified into three areas: algorithmic development, improvements in the efficiency of model evaluation, and hardware-assisted approaches. Early work in the area of algorithmic development included *tim-*

ing analysis [Cha75, New78a, DeM80], which is a simplified form of relaxation-based circuit simulation, and *tearing methods*, which have been applied to both linear [San77, Yan80, Sak81] and nonlinear [Rab79] equation levels to exploit the inactivity of large circuits. More recently, the relaxation-based approaches have been the focus of intensive research. The Waveform Relaxation method [Lel81, Whi83] has been implemented in a number of programs including RELAX [Lel81, Whi83], SWAN [DeMa85], TOGGLE [Hsi85], RealAx [Mar85] and MOSART [Car84]. Iterated Timing Analysis [Kle83, Sal84] has been implemented in SPLICE [Sal82, Kle83] and ELDO [Hen85]. The relaxation-based simulation techniques are the central focus of this dissertation.

Model evaluation is usually associated with the calculation of the current and conductance values for complex devices such as MOS and bipolar transistors. The simulation time in SPICE2 for small and medium size circuits is dominated by model evaluation [New78b]. A number of researchers have attempted to reduce the computation time by using look-up tables for active devices [Cha75, New79, Shi82, Bur83]. In this approach, a number of tables of device characteristics are generated prior to the analysis and simple table look-up operations are performed during the analysis in place of expensive analytical evaluations which often involve the evaluation of many transcendental functions.

In the hardware-assisted approaches, the use of special-purpose microcode to reduce the time required to solve the sparse linear equations has been investigated [Coh81]. Vector processors have also been applied to the matrix solution to exploit the structural regularity and relative inactivity of large circuits [Vla81]. The relaxation-based approaches have been implemented on a number of parallel processors [Deu84, Whi85, Mat85, Web87]. Recently, a special-purpose board for model evaluation has been described [Gyu85]. A simulation engine has also been developed [Auc85] which

uses the timing analysis algorithm of the MOTIS program [Cha75]. In addition, an approach which combines relaxation algorithms and special-purpose hardware has been reported [Whi86].

In this dissertation, a number of new circuit simulation algorithms based on nonlinear relaxation [OrRh70] are presented. These algorithms are analyzed to determine the maximum speed improvement that can be obtained over direct methods under ideal conditions. The ideal speed-up is compared to the actual speed-up of the SPLICE3 program which uses a number of algorithms described in this dissertation. The nonlinear relaxation schemes are also extended to function spaces and applied to the circuit simulation problem. This new approach, called Waveform Relaxation-Newton [Whi85b], and its implementation in the SPLAX program are also described. The simulation techniques in the SPLICE3 program have been implemented on a parallel processor in the PSPLICE program. The results from this program and parallel aspects of other relaxation algorithms are also presented in this dissertation.

In Chapter 2, the transient analysis problem is formulated and the numerical techniques used to solve the problem are described. The linear Gauss-Jacobi (GJ) and Gauss-Seidel (GS) methods are described and their convergence properties are presented. Then the nonlinear relaxation methods, the main focus of this dissertation, are described. The advantages of this approach over linear relaxation are outlined. Finally, the *Waveform Relaxation* (WR) method is described and its convergence properties are presented. The requirement for partitioning to improve the convergence speed of relaxation methods is briefly introduced at the end of the chapter.

The focus of Chapter 3 is time-step control for circuit simulation. Initially, the constraints imposed on the step size by the numerical methods are presented. Next, the issues associated with the implementation of an efficient time-step control scheme are

described. Two properties of waveforms called *latency* and *multirate behavior* are defined and simple experiments are provided to compute upper bounds on the speed improvement if these two properties are exploited under ideal conditions. The efficiency of the relaxation-based techniques in exploiting latency and multirate behavior is also examined.

In Chapter 4, a number of algorithms based on nonlinear relaxation methods are described. A technique which combines nonlinear relaxation [OrRh70] with *event-driven, selective-trace* [SzTh75] to exploit waveform latency is described. This approach is referred to as Iterated Timing Analysis or ITA [Sal82]. Its name is derived from the original work on timing analysis pioneered in the MOTIS program [Cha75]. A preliminary version of ITA was implemented in the prototype mixed-mode simulator SPLICE1.7 [Sal84] and an advanced version in SPLICE2 [Kle84]. A new robust version of ITA has been implemented in the SPLICE3.1 program. The details of the implementation of ITA in all of the above programs are provided in Chapter 4. A number of issues concerning latency and event scheduling are also presented.

The results presented in Chapter 3 provide a strong incentive to exploit the multirate property of circuits. In Chapter 5, a new approach to multirate integration based on the ITA method is described. Initially, a number of previous implementations of event-driven multirate integration schemes and their limitations are described. Then the new multirate ITA scheme is presented. In this new scheme, the basic ITA approach is modified to solve different components in the system using different time-steps by iterating across a "ragged" boundary in time. This method retains the inherent advantage of ITA of relatively inexpensive iterations. In addition, time moves incrementally forward and the components are solved using event-driven techniques. Therefore, it is well-suited for use in mixed-mode simulation programs [New78a, Sak81, Sal83, Kle84]

since they also use event-driven techniques for simulation at higher levels of abstraction. A new method for limiting the effect of a step rejection using a *selective backup* strategy is introduced. Simulation results using this technique are provided.

A waveform-based approach to multirate integration is described in Chapter 6. This approach is based on the WR algorithm and uses a technique called "Waveform-Newton" to solve the iteration equations presented by the Waveform Relaxation method. The combined Waveform Relaxation-Newton (WRN) algorithm can be viewed as a function space extension of the nonlinear relaxation methods used in ITA. The motivation for using the Waveform-Newton method is given and the equations for the method are derived. An iterative step size refinement strategy which improves the accuracy of the numerical integration as the relaxation iterations approach convergence is also described.

In Chapter 7, the parallel aspects of the ITA and WRN methods are explored. The basic concepts of parallel computation are described briefly, followed by a description of the asynchronous computation model. Next, the implementation of ITA on multiprocessors is described, including a description of MSPLICE and a new program called PSPLICE. A novel technique for parallelizing Waveform-Newton is also described. To close the chapter, a generalized space-time model for scheduling is developed as a framework for the analysis of parallel circuit simulation algorithms.

Conclusions and directions for future work are provided in Chapter 8.

CHAPTER 2

RELAXATION-BASED CIRCUIT SIMULATION

2.1. THE CIRCUIT SIMULATION PROBLEM

2.1.1. Equation Formulation

General-purpose circuit simulation programs such as ASTAP [Wee73] and SPICE2 [Nag75] provide a variety of analysis types including DC analysis, time-domain transient analysis, AC analysis, noise analysis and distortion analysis. By far the most CPU-intensive of these analyses is the time-domain transient analysis. The transient analysis problem involves computing the solution of a system of coupled nonlinear differential-algebraic equations over some interval of time, $[0,T]$. The most general form for the equations describing the circuit behavior is:

$$F(\dot{x}(t), x(t), u(t)) = 0 \quad x(0)=X \tag{2.1}$$

where, $x(t) \in \mathbb{R}^n$ is the vector of unknowns, and may be a mixture of node voltages, branch currents, capacitive charges or inductive fluxes, $u(t) \in \mathbb{R}^r$ is a vector of independent sources, $F: \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^r \rightarrow \mathbb{R}^n$, and the initial conditions, $x(0)$, are specified by the vector X.

Equations of this form arise as a result of the properties of general electronic circuits. For example, the current through a capacitor is a function of the time derivative of the voltage across the capacitor and therefore Eqn. (2.1) is dependent on $\dot{x}(t)$. Since many devices have nonlinear relationships between their currents and voltages, F is also usually nonlinear. And finally, as a circuit is constructed from a collection of sparsely connected elements, F is a sparse function of the components of x. These circuit properties all have some impact on the numerical techniques used to solve the tran-

sient simulation problem, and the resulting efficiency with which the solution is obtained.

There are a number of different ways to formulate the circuit equations described by Eqn. (2.1). The most popular of these are Nodal Analysis (NA) [DeKu69], Modified Nodal Analysis (MNA) [Ho75] and Sparse Tableau Analysis (STA) [Hac71]. These formulations are all based on the application of Kirchoff's Current Law (KCL), Kirchoff's Voltage Law (KVL) and the branch constitutive equations [DeKu69]. Nodal Analysis is the simplest of the three approaches. It uses KCL, which requires that the sum of the currents entering each node equals the sum of the currents leaving each node. In a circuit containing $n+1$ nodes, if KCL is written for every node in the circuit, a system of n equations is obtained assuming that one node is defined as a reference node. The currents in each equation can be replaced with the branch constitutive relations which are functions of the branch voltages (by assumption in NA), and KVL can be used to replace the branch voltages by node voltages. KVL requires that the sum of the voltages around any loop in a circuit be identically zero. The n node voltages are the unknown variables in this formulation. Note that it must be possible to represent the element and input source currents in terms of their terminal voltages in order apply Nodal Analysis. This requirement excludes current-controlled devices, floating voltage sources[1] and inductors and therefore limits the scope of the NA technique. However, inductors and floating voltage sources can be included in NA by simply reorganizing their branch equations as described in [McC75, Whi85c]. Since the other current-controlled devices are not frequently used in the simulation of integrated circuits, NA is an adequate formulation technique for most practical circuits.

[1] These are voltage sources with neither terminal connected to the ground node.

The formulation used throughout the rest of this dissertation is Nodal Analysis. The NA equations are formulated as follows: First, KCL is applied at each node in a circuit with n nodes and b branches to produce a matrix equation of the form:

$$A\, i = 0 \tag{2.2}$$

where $A \in \mathbb{R}^{n \times b}$ is the reduced incidence matrix with entries of either +1, -1 or 0 and $i \in \mathbb{R}^b$ is the vector of branch currents in the circuit. Element a_{ik} of A is +1 if a particular branch current, i_k, leaves node i, -1 if it enters node i and 0 if it is not incident at node i. If the set of branch currents are divided into the capacitor currents, i_c, and the currents through the resistive elements, i_r, then Eqn. (2.2) can be rewritten as:

$$A_c\, i_c = -A_r\, i_r \tag{2.3}$$

where $A = [A_c \mid A_r]$ and $i = [i_c, i_r]^T$.

Each of the currents due to the nonlinear resistive elements can be replaced by their branch constitutive relations which are all functions of the branch voltages by assumption. The branch voltages, v_b, can be replaced by the node-to-datum voltages, v, using the relation:

$$A^T v = v_b \tag{2.4}$$

which follows from KVL [ChLi75]. Then, the right-hand side of (2.3) can be written as:

$$A_r\, i_r = -\begin{bmatrix} f_1(v) \\ f_2(v) \\ \cdot \\ \cdot \\ f_n(v) \end{bmatrix} \tag{2.5}$$

where $f_k(v)$ is the sum of all the currents through the resistive elements connected to node k as a function of the node voltages, v.

The left-hand side of Eqn. (2.3) represents the capacitor currents. The nonlinear capacitors are often specified in terms of their stored charge, q, as function of the voltage across the capacitor, v_c, as follows:

$$q = q(v_c)$$

The current flowing through the capacitor can be obtained by taking the time-derivative of charge, which can then be related to the capacitance by applying the chain-rule:

$$i_{cap} = \dot{q}(v_c) = \frac{dq(v_c)}{dv_c}\frac{dv_c}{dt} = C(v_c)\dot{v}_c \tag{2.6}$$

Hence, each of the components of i_c in Eqn. (2.3) can be replaced by $C(v_c)\dot{v}_c$. If Eqn (2.4) is used to replace the branch voltages by node voltages, then $A_c i_c$ can be transformed into the following:

$$A_c i_c = \begin{bmatrix} C_{11}(v) & \cdots & C_{1n}(v) \\ \vdots & \ddots & \vdots \\ C_{n1}(v) & \cdots & C_{nn}(v) \end{bmatrix} \begin{bmatrix} \dot{v}_1 \\ \vdots \\ \dot{v}_n \end{bmatrix} \tag{2.7}$$

An important assumption which is used to guarantee convergence of relaxation-based simulation techniques (to be described shortly) is that a two-terminal capacitor exists between each node and some reference node. These are referred to as *grounded capacitors*. This assumption is easily satisfied by circuits where lumped, parasitic capacitances are present between circuit interconnect and ground or on the terminals of active circuit elements. Therefore, in the capacitance matrix above, all C_{ii}'s are non-zero. Note that C_{ij} is zero if a capacitor does not exist between nodes i and j in the circuit.

By combining Eqns. (2.5) and (2.7), one obtains:

$$\begin{bmatrix} C_{11}(v) & \cdots & C_{1n}(v) \\ \vdots & \ddots & \vdots \\ C_{n1}(v) & \cdots & C_{nn}(v) \end{bmatrix} \begin{bmatrix} \dot{v}_1 \\ \vdots \\ \dot{v}_n \end{bmatrix} = - \begin{bmatrix} f_1(v) \\ \vdots \\ f_n(v) \end{bmatrix} \tag{2.8}$$

This equation can be written in the compact form:

$$C(v(t),u(t))\,\dot{v}(t) = -f(v(t),u(t)), \quad t \in [0,T]. \tag{2.9}$$

$$v(0) = V.$$

where $v(t) \in \mathbb{R}^n$ is the vector of node voltages at time t, $\dot{v}(t) \in \mathbb{R}^n$ is the vector of time derivatives of $v(t)$, $u(t) \in \mathbb{R}^r$ is the input vector at time t, $C(x(t),u(t))$

represents the nodal capacitance matrix, and:

$$f(v(t),u(t))=[f_1(v(t),u(t)),\cdots,f_n(v(t),u(t))]^T$$

where $f_k(v(t),u(t))$ is the sum of the currents charging the capacitors connected to node k.

Eqn. (2.9) is a set of coupled first-order nonlinear differential equations which uses voltage as a state variable. This is commonly referred to as the capacitance formulation of the transient analysis problem. Alternatively, charge may be used as a state variable rather than voltage. The proper choice of voltage or charge as the state variable depends on the nature of the capacitors in the circuit. If all capacitances are linear, then either voltage or charge may be used as the state variable. However, in circuits with nonlinear capacitors, such as MOS circuits, charge must be used as the state variable due to considerations of charge conservation [War, Yan83, Whi85c]. That is, in order to keep the total charge in the system constant during the simulation process, charge must be used as the state variable. Examples of charge conservation problems arising from the use of Eqn. (2.9) are given in [War78, Yan83, Whi85c].

The charge formulation of the circuit equations in normal form is given by:

$$\dot{q}(t)=i(q(t))$$

where $q_k(v)$ is the sum of the charges due to the capacitors connected to node k and $i_k(q)$ is the sum of the currents charging the capacitors at node k. This equation can be solved to obtain the node charges as a function of time. However, information about charge is of little interest to the circuit designer. The designer would prefer to have information about the node voltages from the simulator. Therefore, it is preferable to write the charge formulation as

$$\dot{q}(t)=i(q(t))=-f(v(t))$$

which is obtained by combining Eqn. (2.6) and Eqn. (2.9). This assumes that q is an

invertible function of v. The charge formulation, including the input sources, $u(t)$, is given by:

$$\dot{q}(v(t),u(t)) = f(v(t),u(t)). \tag{2.10}$$

Both the formulations given by Eqns. (2.9) and (2.10) will be used throughout this dissertation.

2.1.2. Numerical Techniques for Transient Analysis

Eqns. (2.9) and (2.10) formulated above for the transient analysis of circuits must be solved using numerical techniques since, in general, it is difficult to obtain closed-form solutions. The first step is to apply a numerical integration method to discretize the time-derivative, $\dot{x}(t)$. An integration method divides the continuous interval of time, $[0,T]$, into a set of M discrete time points defined by:

$$t_0=0 \,, \quad t_{n+1} = t_n + h_n \,, \quad t_M = T. \tag{2.11}$$

An algebraic problem is solved at each time point, t_{n+1}, to obtain a sequence approximation to the exact solution. The quantity h_n is referred to as a time-step. The selection of proper time-steps for a given problem is an important issue which is described in detail in Chapter 3. An example of a first-order implicit integration method is the backward-Euler (BE) method. To solve $\dot{x}(t)=f(x(t))$ using BE, the following expression is used:

$$x(t_{n+1}) = x(t_n) + h_n f(x(t_{n+1})) \tag{2.12}$$

This equation is implicit in that $x(t_{n+1})$ appears on both sides of the equation.

A numerical integration method converts a set of nonlinear differential equations into a set of nonlinear algebraic equations. These algebraic equations must be solved using some numerical method at each time point. The most commonly used method to solve nonlinear equations is the Newton-Raphson method [OrRh70]. To solve a system of nonlinear equations, given by $F(x)=0$, using the Newton-Raphson method, the

following iterative equation is used:

$$J_F(x^k)(x^{k+1} - x^k) = -F(x^k) \tag{2.13}$$

where $J_F(x)$ is the Jacobian matrix and k is the iteration counter for the method. Each term in the Jacobian matrix, g_{ij}, is given by:

$$g_{ij} = \frac{\partial F_i}{\partial x_j} \tag{2.14}$$

where F_i is the i th component of F and x_j is the j th component of x. Eqn. (2.13) is iterated until $||x^{k+1}x^k|| < \epsilon_1$ and $||F(x^{k+1})|| < \epsilon_2$. Note that if the problem is linear, then the Newton method produces the correct solution in one iteration.

The Newton method described above converts the set of coupled nonlinear algebraic equations into a set of coupled linear equations given by $Ax = b$, where $x \in \mathbb{R}^n$, $b \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$ and A is assumed to be nonsingular. The matrix A is relatively sparse, typically having three elements per row [NeSa83]. There are essentially two approaches to solving a sparse linear system. One approach is to use *direct methods* (such as LU decomposition) which attempt to exploit the sparse nature of the matrix during the computation. The implementation of these methods involves carefully chosing a data structure and the use of special pivoting strategies to minimize fillins [Kun86]. A second approach to the sparse linear problem is to use *relaxation methods*. The relaxation process involves decoupling the system of equations and solving each equation separately. An iterative method is applied between the equations until convergence is obtained. In effect, the problem of solving one large system containing n variables is converted to the problem of solving n subsystems each containing one variable.

2.1.3. Overview of Standard Circuit Simulation

The standard approach to circuit simulation is based on direct methods and uses the following steps:

1. MNA is used to formulate the system of differential-algebraic equations for the circuit.

2. Implicit integration methods are applied to convert the differential equations into a sequence of algebraic equations, which are nonlinear in general.

3. A damped Newton-Raphson method is used to convert the nonlinear equations into linear equations.

4. Direct sparse-matrix techniques are used to solve the linear equations generated by the Newton-Raphson method.

The details of the implementation of this approach in SPICE2 may be found in [Nag75]. This approach has proven to be very reliable and can be used across a variety of different technologies and element types. The most computationally intensive part of this approach is the Newton-Raphson iteration. It is composed of two phases: the formulation phase and the solution phase. These two phases, represented by steps 3 and 4 above, are repeated at each time point until convergence is obtained. In the formulation phase, the elements in the circuit are processed by calculating their contribution to the Jacobian matrix and the right-hand side vector in Eqn. (2.13) to form the system of linear equations. This is also referred to as the function evaluation (or model evaluation) and load phase, and can be very time-consuming because of the complexity of the equations describing the elements in the circuit. For small to medium size circuits containing MOS devices, the model evaluation and load times dominate the total CPU-time for the simulation [New78].

In the second phase of the Newton iteration, the linear equations generated in the first phase are solved using direct methods such as LU decomposition. While this portion has a negligible contribution to the total run time for small circuits, it can in fact

dominate the run time for very large circuits (i.e. greater than 1000 nodes in the circuit for SPICE2) [NeSa83]. Therefore, any technique which attempts to reduce overall circuit simulation run times must reduce both the model evaluation time and the linear equation solution time to be effective.

2.2. RELAXATION METHODS

Relaxation-based circuit simulators, such as SPLICE [Sal83, Kle84] and RELAX [Lel81, Whi83], use iterative methods at some stage of the solution process to solve the circuit equations. The success of these programs is due to the fact that they offer the same level of accuracy as direct methods, assuming identical device models, while significantly reducing the overall simulation run time. The reduction in run time is accomplished by computing fewer solution points for each waveform, thereby reducing the total number of model evaluations, and by avoiding the direct sparse-matrix solution. However, a tradeoff exists in the relaxation methods since they can only be applied to a specific class of circuits. Furthermore, there is the additional requirement that a grounded capacitor be present at each node in the circuit to guarantee convergence. While these factors limit the scope of the application of relaxation methods, the programs which use relaxation have proven to be extremely useful for simulation of many industrial MOS and Bipolar integrated circuits. In the remainder of this chapter, the relaxation methods are described and their mathematical properties are presented.

2.2.1. Linear Relaxation

Two common linear iterative methods are the Gauss-Jacobi (GJ) and Gauss-Seidel (GS) methods. The methods differ only in the information they use when solving a particular equation as shown in the two algorithms given below. The superscript k is the iteration count, and ϵ is some small error tolerance.

Algorithm 2.1 (Gauss-Jacobi Method to solve $Ax = b$)

$k \leftarrow 0$;
guess some x^0 ;
repeat {
$k \leftarrow k+1$;
forall ($i \in \{ 1, \cdots, n \}$)

$$x_i^k = \frac{-1}{a_{ii}} \left[\sum_{j=1}^{i-1} a_{ij} x_j^{k-1} + \sum_{j=i+1}^{n} a_{ij} x_j^{k-1} \right] ;$$

} **until** ($|x_i^k - x_i^{k-1}| \leqslant \epsilon, i = 1, \cdots, n$);

■

Algorithm 2.2 (Gauss-Seidel Method to solve $Ax = b$)

$k \leftarrow 0$;
guess some x^0 ;
repeat {
$k \leftarrow k+1$;
foreach ($i \in \{ 1...n \}$)

$$x_i^k = \frac{-1}{a_{ii}} \left[\sum_{j=1}^{i-1} a_{ij} x_j^k + \sum_{j=i+1}^{n} a_{ij} x_j^{k-1} \right] ;$$

} **until** ($|x_i^k - x_i^{k-1}| \leqslant \epsilon, i = 1, \cdots, n$);

■

Notice that in the GJ method each x_i^k is computed using the iteration values $x_j^{(k-1)}$, $j = 1, \cdots, n$, which are the values from the previous iteration. In the GS method, the latest iteration values are used as soon as they become available. The **forall** construct in Algorithm 2.1 suggests that all n variables can be computed in parallel during each iteration. The **foreach** construct in Algorithm 2.2 requires that the variables be processed in a particular sequence.

Linear relaxation schemes are usually described using a *splitting* notation that separates A into two components:

$$A = B - C \tag{2.15}$$

where B is a nonsingular matrix such that linear systems of the form $Bx = d$ are "easy" to solve. Various relaxation schemes can be constructed by setting B and C in the iterative equation:

$$x^{k+1} = -B^{-1}Cx^{k} + C^{-1}b$$

In particular, if A is decomposed into its diagonal, strictly- lower-triangular and strictly-upper-triangular parts, D, L and U, respectively such that $A = L + D + U$, then the GS method is obtained by setting

$$B = (L+D) \qquad C = -U \tag{2.16}$$

and the GJ method is obtained using

$$B = D \qquad C = -(L+U). \tag{2.17}$$

Since relaxation methods are iterative, the question naturally arises as to whether or not these methods converge to the correct solution and, if so, under what conditions? The requirements for convergence are stated in the following standard theorem [Var61]:

Theorem 2.1: Suppose $b \in \mathbb{R}^n$ and $A = B - C \in \mathbb{R}^{n \times n}$ is nonsingular. If B is non-singular and the spectral radius of $B^{-1}C$, given by $\rho(B^{-1}C)$, satisfies the condition $\rho(B^{-1}C) < 1$, then the iterates $x^{(k)}$ defined by $Bx^{(k+1)} = Cx^{(k)} + b$ converge to $x = A^{-1}b$ for any starting vector $x^{(0)}$. ■

In other words, the magnitude of the largest eigenvalue of the iteration matrix $B^{-1}C$ must be strictly less than 1 to guarantee convergence of a linear relaxation method. A condition which guarantees that $\rho(B^{-1}C) < 1$ is if A is strictly diagonally dominant. A matrix has this property if the diagonal term in each row i is greater than the sum of the off-diagonal terms in the same row, i.e.,

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| < |a_{ii}| \quad \text{for } 1 \leq i \leq n$$

and the "more dominant" the diagonal, the more rapid will be the convergence.

A number of techniques are available to improve the convergence speed of linear relaxation methods. For example, in the GS method, the order in which the equations are solved usually has a strong affect on the number of iterations required to converge.

Consider the case when matrix A is lower triangular. If processed in the sequence, $x_1, x_2, \cdots, x_n$, then one relaxation iteration is sufficient to obtain the correct solution. However, if processed in the reverse order, then n iterations are required to obtain the solution. Therefore, equation ordering is usually performed on the variables whenever GS is used. Techniques for equation ordering are described in the chapters to follow.

Another technique to improve convergence, also used in conjunction with the Gauss-Seidel method, is the method of Successive Overrelaxation (SOR). In this approach, the Gauss-Seidel method is used initially to generate an intermediate value, $\tilde{x}_i^{(k+1)}$, using the equation

$$\tilde{x}_i^{(k+1)} = B^{-1}Cx_i^{(k)} + B^{-1}b$$

where B and C are defined by Eqn. (2.16). The actual value of $x_i^{(k+1)}$ is obtained by taking a weighted combination of the previous iteration and the intermediate value which depends on a relaxation parameter, ω.

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \omega\tilde{x}_i^{(k+1)}$$

The SOR method can also be defined in terms of the splitting notation with $B = \omega^{-1}(D + \omega L)$, and $C = \omega^{-1}[(1-\omega)D - \omega U]$. While the proper choice of ω can greatly reduce the number of iterations, an optimal value of ω can only be computed *a priori* for a limited number of cases. In general, it may be necessary to perform a somewhat complicated eigenvalue analysis to determine the best value of ω. In practice, adaptive algorithms are used to select an appropriate value for ω during the solution process.

Linear relaxation methods can be used in conjunction with the solution of nonlinear equations to solve the linear systems generated by Newton's method. For example, the Newton-SOR method is a combination of the Newton-Raphson method and the SOR method. In this composite algorithm, the Newton iteration can be considered as the

"outer loop" and the SOR iteration as the "inner loop". While it is possible to carry the inner loop to convergence, there no requirement to do so, as long as the outer loop is iterated to convergence. In general, an m-step Newton-SOR method can be defined where m is the number of iterations used in the inner loop. For the case $m=1$, a one-step Newton-SOR method is obtained. The Newton-SOR method is only one example of the possible combinations of nonlinear iterative methods and linear iterative methods. For example, Newton's method may be replaced by the secant method and the SOR iteration may be replaced by one of the standard Gauss-Seidel or Gauss-Jacobi methods.

2.2.2. Nonlinear Relaxation

The basic idea of relaxation can also be extended to solve systems of nonlinear equations of the form $F(x) = 0$, where $F:\mathbb{R}^n \rightarrow \mathbb{R}^n$, with components $f_1, f_2, \cdots, f_n$ and $f_i:\mathbb{R}^n \rightarrow \mathbb{R}$. That is, rather than solving the system using direct matrix techniques, the nonlinear equations can be solved in a decoupled fashion. Two such algorithms are given below. The index k is the iteration count, while ϵ_1 and ϵ_2 are error tolerances.

Algorithm 2.3 (Nonlinear Gauss-Jacobi Method to solve $F(x) = 0$)

$k \leftarrow 0$;
guess some x^0;
repeat {
  $k \leftarrow k+1$;
  forall ($i \in \{1, \cdots, n\}$)
    solve $f_i(x_1^{k-1}, \cdots, x_{i-1}^{k-1}, x_i^k, x_{i+1}^{k-1}, \cdots, x_n^{k-1})=0$ for x_i^k;
} **until** ($|x_i^k - x_i^{k-1}| \leq \epsilon_1, |f_i(x^{k,i})| \leq \epsilon_2, i=1, \cdots, n$);

■

Algorithm 2.4 (Nonlinear SOR Method to solve $F(x) = 0$)

$k \leftarrow 0$;
guess some x^0 ;
repeat {
$k \leftarrow k+1$;
foreach ($i \in \{1, \cdots, n\}$)
solve $f_i(x_1^k, \cdots, x_{i-1}^k, x_i^k, x_{i+1}^{k-1}, \cdots, x_n^{k-1}) = 0$ for x_i^k ;

$x_i^k \leftarrow (1-\omega)x_i^k + \omega(x_i)$;

} **until** ($|x_i^k - x_i^{k-1}| \leqslant \epsilon_1$, $|f_i(x^{k,i})| \leqslant \epsilon_2, i = 1, \cdots, n$);

■

where $x^{k,i} = (x_1^k, \cdots, x_{i-1}^k, x_i^k, x_{i+1}^{k-1}, \cdots, x_n^{k-1})$.

These algorithms are referred to as nonlinear relaxation methods. The steps are very similar to linear relaxation as given in Algorithms (2.16) and (2.17) except that, in this case, each equation in the inner loop is nonlinear. To solve each one-dimensional nonlinear problem, $f_i(x) = 0$, an iterative technique such as the Newton method or secant method must be used since, in general, a closed-form solution cannot be obtained. Combining the SOR method with the Newton method results in the SOR-Newton algorithm. The general case is the m -step SOR-Newton method, where m is the number of Newton iterations taken in the inner loop. The question again arises as to the number of inner loop iterations to use.

It can be shown that the rate of convergence of the one-step SOR-Newton method is the same as the one-step Newton-SOR method [OrRh70]. The m -step SOR-Newton method also has the same rate as the one-step method implying that it is not worthwhile to take more than one Newton step since the convergence rate is not affected. However, the convergence rate of the m -step Newton-SOR method is m times the rate of convergence of the one-step method. Therefore, based on the rates of convergence, one might be inclined to choose the m -step Newton-SOR to solve a system of nonlinear equations. There is however a hidden cost if the partial derivatives are expensive to calculate. Each step of SOR-Newton requires the evaluation of each f_i

and n partial derivatives, $\frac{\partial f_i}{\partial x_i}$. whereas the m-step Newton-SOR method requires the evaluation of f and all partial derivatives. Based on both operation counts and the rates of convergence given above, the one-step SOR-Newton method appears to be the most efficient and for this reason it is used in Iterated Timing Analysis (ITA) [Sal83]. Note that this implies one iteration in the inner loop. The outer loop is iterated until convergence is obtained. SOR-Newton also offers one additional advantage over Newton-SOR in that waveform latency can be exploited easily and this feature is described in more detail in the chapters to follow.

In a general-purpose implementation of these methods, the iterative process must be terminated when the solution is close enough to x^*. Often, this condition is checked using the test $|x_i^{k+1} - x_i^k| \leq \epsilon_1$. However, this check of convergence is not sufficient in the nonlinear case. A second test is necessary to ensure that each function, f_i, is close enough to zero and this is specified using the test $|f_i(x^{k+1,i})| \leq \epsilon_2$ for all i.

The algorithms presented above are meaningful only if the nonlinear equations, which are solved at each step in the inner loop, have unique solutions in some specific domain under consideration. Recall that for linear relaxation, the condition that $a_{ii} \neq 0$, for all $i = 1, \cdots, n$ ensures that a solution exists, assuming that the diagonal dominance property holds. A similar condition is required in the nonlinear case. To illustrate this point, let the Jacobian be decomposed into its diagonal, strictly lower-triangular and strictly upper-triangular parts as follows:

$$F'(x) = D(x) + L(x) + U(x)$$

The iterations in the nonlinear scheme are well-defined if F is continuously differentiable in an open neighborhood S of the point x^*, for which $F(x^*) = 0$, and $D(x^*)$ is nonsingular. The requirements for convergence are also analogous to the linear case. By splitting the Jacobian matrix using the previous notation

$$F'(x) = B(x) - C(x),$$

the local convergence of the nonlinear relaxation methods described in Algorithms (2.5) and (2.6) can be stated as follows [OrRh70]:

Theorem 2.2: Given $F:\mathbb{R}^n \rightarrow \mathbb{R}^n$, assume that F is continuously differentiable in an open neighborhood S of x^* and x^* satisfies $F(x^*)=0$. If $B(x^*)$ is nonsingular and $\rho(B(x^*)^{-1}C(x^*))<1$, then there exists an open ball $S^* \subset S$ such that the nonlinear relaxation methods given in Algorithms (2.5) and (2.6) converge to x^* for any initial guess $x^0 \in S^*$. ■

Recall that under the conditions stated in Theorem 2.1, linear relaxation methods converge for any initial guess. However, for the nonlinear case the convergence result is local since the initial guess must be close enough to the final solution to guarantee convergence. The proof of this theorem may be found in the reference [OrRh70].

2.2.3. Waveform Relaxation

The relaxation schemes presented above can be also extended to functions spaces to solve systems of differential equations. This class of algorithms is called *Waveform Relaxation* (WR) [Lel81]. The relaxation variables in WR are elments of function spaces, i.e., they are waveforms in the closed interval $[0,T]$, whereas for linear and nonlinear relaxation the variables are simply vectors in Euclidean n-space. To illustrate the WR algorithm, consider the circuit simulation problem in the form specified in Eqn. (2.9). The WR algorithm for solving this system of equations is as follows:

Algorithm 2.5 (WR Gauss-Seidel Algorithm for solving Eqn. (2.9))

$k \leftarrow 0$:
guess waveform $x^0(t) : t \in [0,T]$ such that $x^0(0) = x_0$:
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** ($\max_{1 \leq i \leq n} \max_{t \in [0,T]} | x_i^k(t) - x_i^{k-1}(t) | \leq \epsilon$)
■

Algorithm 2.5 converts the problem of solving a coupled system of n first-order ODE's to the problem of solving n separate differential equations, each containing a single variable. The outer loop in the algorithm is the Gauss-Seidel iteration which requires that the latest values of the relaxation variables be used to solve each equation in the inner loop. The inner loop equations are single differential equations each of which is solved using some numerical integration method. The convergence of the Waveform Relaxation method is guaranteed under conditions which are similar to the linear and nonlinear cases, as stated in the following theorem [Whi85c]:

Theorem 2.3: If $C(x(t), u(t)) \in \mathbb{R}^{n \times n}$ of Eqn. (2.9) is strictly diagonally dominant uniformly over all $x(t) \in \mathbb{R}^n$ and $u(t) \in \mathbb{R}^r$ and Lipschitz continuous with respect to $x(t)$ for all $u(t)$, then the sequence of waveforms $\{x^k\}$ generated by the Gauss-Seidel or Gauss-Jacobi WR algorithm will converge uniformly to the solution of Eqn. (2.9) in any bounded interval $[0,T]$, for any initial guess $x^0(t)$. ■

While this theorem guarantees convergence of the WR algorithm, it does not imply anything about the speed of convergence. Although the method usually converges in a few iterations, it has been observed that in test cases with tight feedback loops, the number of iterations required to converge is proportional to the simulation interval [Whi83]. To improve convergence, the simulation interval $[0,T]$ is usually divided into smaller intervals, $[0,T_1]$, $[T_1,T_2]$, ... , $[T_{n-1},T_n]$, called *windows*. Initially, the WR algorithm is applied only in the first window, $[0,T_1]$, until the waveforms converge. Then a second window, $[T_1,T_2]$, is selected and WR is applied within this interval until the waveforms converge. This continues until the entire simulation interval is covered. Note that the WR method converges more rapidly as the window size is made smaller. One advantage of WR is that the time-steps for each of the variables can be chosen independent of one another but this advantage is compromised if the windows are too small. Therefore, the window size is an important factor which determines the performance of programs which use the WR method.

2.2.4. Partitioning for Relaxation Methods

Relaxation methods are most effective when applied to a system of equations which are "loosely-coupled", that is, where the variables do not depend too strongly on one another. For this type of system, relaxation methods usually converge quite rapidly. The speed of convergence in the linear case is controlled by the spectral radius of the iteration matrix given by $\rho(B^{-1}C)$ (using the notation of Theorem 2.1) and this is usually close to zero for loosely-coupled systems. However, for an arbitrary problem, there is no guarantee that the spectral radius will be small. In fact, in "tightly-coupled" systems, the spectral radius may be very close to 1 which implies slow convergence. This degrades the performance of the relaxation-based methods compared to the direct methods.

The precise meaning of loosely-coupled and tightly-coupled can be described using a simple 2x2 matrix problem:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Assume that the equations have been ordered such that x_1 is solved before x_2. Then, a_{21} can be considered as a feed-forward term and a_{12} can be considered as a feedback term. The spectral radius of the iteration matrix for the GS method (see Eqn. 2.1) is given by:

$$\rho(B^{-1}C) = \left| \frac{a_{12}a_{21}}{a_{11}a_{22}} \right|$$

and to guarantee convergence, this value must be strictly less than 1. If both a_{12} and a_{21} are non-zero, the variables x_1 and x_2 are considered to be coupled. If both a_{12} and a_{21} are large, relative to a_{11} and a_{22}, then x_1 and x_2 are called *tightly-coupled* variables. If both a_{12} and a_{21} are small, then x_1 and x_2 are called *loosely-coupled* variables. Note that if either a_{21} or a_{12} is zero, then equation ordering has a significant impact on the number of iterations. In fact, if a_{21}=0, then x_2 should be solved before x_1 so that the solution can be obtained in one iteration. A similar argument applies if a_{21} is very small compared to a_{12}. Therefore, the main objective in reordering is to make the A matrix as lower triangular as possible.

When solving large systems, the definitions given above can be used to partition the system into groups of tightly-coupled variables. Rather than using relaxation methods to solve the tightly-coupled variables within each "block", it is better to solve them using direct methods. The relaxation method can be applied between the blocks, which are loosely-coupled relative to the variables within a block. This gives rise to block relaxation methods [Var61], which can be viewed as a combination of the direct

methods and relaxation methods. As an example, consider the 3x3 matrix problem:

$$\begin{bmatrix} a_{11} & a_{12} & 0 \\ 0 & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

If x_2 and x_3 are tightly-coupled, then many relaxation iterations may be required to solve this problem. However, by grouping x_2 and x_3 into the same block and reordering the variables for the Gauss-Seidel method, the following equation is obtained:

$$\begin{bmatrix} a_{22} & a_{23} & 0 \\ a_{32} & a_{33} & 0 \\ a_{12} & 0 & a_{11} \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \\ x_1 \end{bmatrix} = \begin{bmatrix} b_2 \\ b_3 \\ b_1 \end{bmatrix}$$

If x_2 and x_3 are solved using direct methods, then this problem can be solved using a single relaxation iteration. This example shows that proper ordering and partitioning are extremely important in the relaxation-based methods. The main problem, of course, is to find a suitable partitioning of the variables. A number of schemes to perform this function are described in the chapters to follow. In addition, the tradeoff between the number of partitions (and hence the available parallelism) and the number of iterations required for convergence in the nonlinear relaxation process is considered in Chapter 7.

CHAPTER 3

EFFICIENT TIME-STEP CONTROL FOR CIRCUIT SIMULATION

3.1. INTRODUCTION

Time-step control is an important theme throughout this dissertation. In this chapter, the constraints imposed by the numerical techniques on the step sizes used in the integration process are described. Based on these constraints, an efficient time-step control scheme is presented. A way to further improve the efficiency by using different step sizes to solve different components in the system is also presented. Upper bounds on the speed improvement that can be obtained by solving the equations in this manner are given along with an indication of what factors control the ability to achieve the specified bounds if relaxation methods are used solve the circuit equations.

The circuit simulation problem, in its most general form, involves the solution a system of nonlinear algebraic-differential equations. To simplify the description to follow, the circuit equations are assumed to be a system of differential equations in *normal form*:

$$\dot{x}(t) = f(x(t), u(t)), \quad x(0) = X, \quad t \in [0, T] \tag{3.1}$$

where u is the set of primary inputs, x is a vector of unknown circuit variables and f is some nonlinear function. The vector of values specified as X are the initial conditions, and the simulation interval is $[0, T]$.

As described earlier, the standard approach to solving Eqn. (3.1) is to use a numerical integration method. The goal is to generate the solution as efficiently as possible while providing the desired level of accuracy. At first glance, one may attempt to make the integration process efficient by minimizing the total number of time points used. That is, at any stage during the simulation, take the largest step possible that

provides the required accuracy. This strategy is effective for linear problems, assuming that the numerical integration method has guaranteed stability properties. However, for nonlinear problems, it may be more efficient to take smaller steps so that the iterative method used to solve the nonlinear algebraic equations converges in fewer iterations. Using small time-steps also improves the accuracy of the solution.

A similar situation exists if relaxation is used in the solution process. That is, the iterative process converges more rapidly and the solution is more accurate if smaller steps are used. Therefore, minimizing the total number of iterations used would seem to be a better way to reduce the amount of computation. The cost of each iteration is proportional to the number of model evaluations[1] performed. Therefore, the number of model evaluations used in the solution process is a good measure of the amount of computation used, assuming that the linear equation solution time is small. Based on this argument, a major objective for the efficient solution of the differential equations in Eqn. (3.1) should be to minimize the total number of model evaluations.

A number of researchers have attempted to reduce the computation time for model evaluation by using lookup tables for active devices [Cha75, New79, Shi82, Bur83, Gyu86]. In this approach, a number of tables of device characteristics are generated prior to the analysis, and simple table look-up operations are performed during the analysis in place of the expensive analytic evaluations. Points which are not available in the tables are interpolated using polynomial interpolation or splines. One drawback of this approach is that there may be a substantial memory requirement for these tables, depending on the level of accuracy desired, but it is usually justified by the improvement in computation speed. Current research in this area involves reducing the memory requirements without sacrificing either the computational advantage or the

[1] A model evaluation usually refers to the calculation of the currents and conductances for a MOS or bipolar transistor, or some equivalent amount of computation.

accuracy of the device models. Further details on this topic may be found in the references listed above.

In this chapter, the factors involved in selecting appropriate time-steps to reduce the number of model evaluations are examined. In Section 3.2, the constraints imposed on the step size by the numerical techniques are presented. The issues associated with the implementation of an efficient time-step control scheme are described in Section 3.3. In Section 3.4, waveform latency and multirate behavior are defined and simple experiments are given to compute upper bounds on the speed improvement that can be obtained if these two properties are exploited. In Section 3.5, the efficiency of the relaxation-based techniques in exploiting latency and multirate behavior is examined, and the computational effort needed to produce a solution is estimated.

3.2. CONSTRAINTS ON STEP SIZE

3.2.1. Numerical Integration Method

The general form of a k^{th}-order linear multistep integration method [ShGo75, Gea71] is given by:

$$x_{n+1} = \sum_{i=0}^{p} a_i x_{n-i} + \sum_{i=-1}^{p} h_{n-i-1} b_i \dot{x}_{n-i} \tag{3.2}$$

where x_n is the computed solution at time t_n, and h_n is the time-step at time t_n. The $2p+3$ coefficients, a_i and b_i, are chosen such that Eqn. (3.2) will give the exact solution if the true solution is a polynomial in t of degree less than or equal to k [ChLi75].

There are two broad classes of integration methods: explicit and implicit methods[2] [ChLi75]. Explicit methods use only the solutions at previous time points to generate the solution at the next time point, and are characterized by $b_{-1}=0$. A number of

[2] Recently, a number of combined integration-relaxation methods used in Timing Simulation [Cha75] have been classified as semi-implicit integration methods [New79, DeM80, NeSa83, Whi85c].

explicit integration methods can be derived directly from a Taylor series expansion of $x(t)$ at the point t_n:

$$x_{n+1} = x_n + h_n \dot{x}_n + \frac{h_n^2}{2}\ddot{x}_n + \cdots \tag{3.3}$$

For example, the Forward-Euler (FE) method is obtained by taking the first two terms of Eqn. (3.3):

$$x_{n+1} = x_n + h_n \dot{x}_n \tag{3.4}$$

This *difference* equation can be formulated in terms of Eqn. (3.3) by setting $p=0$, $a_0=1$, $b_0=1$ and all other coefficients to zero. Eqn. (3.4) implies that each equation can be updated independently, and in parallel, at each time point. For differential equations in the normal form, the solution at each time point can be obtained in one step and does not involve a matrix solution, and therefore the explicit methods are extremely efficient. Unfortunately, these methods are not as useful as implicit methods for circuit simulation. Implicit methods are characterized by $b_{-1} \neq 0$ in Eqn. (3.2). The Backward-Euler (BE) implicit integration method can be derived using a Taylor expansion of $\dot{x}(t)$ about the point t_n:

$$\dot{x}_{n+1} = \dot{x}_n + h_n \ddot{x}_n + \frac{h_n^2}{2}\frac{d^3 x_n}{dt^3} + \cdots \tag{3.5}$$

Using Eqn. (3.5) to replace $\dot{x}_n$ in Eqn. (3.3), and ignoring the higher-order terms, the BE scheme is obtained:

$$x_{n+1} = x_n + h_n \dot{x}_{n+1} \tag{3.6}$$

In this case, $p=0$, $a_0=1$, $b_{-1}=1$ with all other coefficients equal to zero. For nonlinear problems, this implicit equation is usually solved using an iterative method, often requiring a matrix solution. Therefore, the implicit methods are computationally more expensive than explicit methods. The Forward-Euler and Backward-Euler methods are representative of their respective class of integration algorithms and will be used to illustrate a number of other properties below.

a. Accuracy Constraint

Integration methods provide a numerical approximation to the true solution since, in general, the exact solution of Eqn. (3.1) cannot be obtained. The error in the numerical solution is due to a combination of the machine error and the truncation error. The machine error is usually in the form of a round-off error, since finite precision arithmetic is used, and it depends on the floating-point arithmetic unit of the computer being used. The truncation error results from the fact that the Taylor series is truncated after a number of terms and this error depends on the specific integration method. The *local truncation error* (LTE) for general multistep methods is defined as:

$$LTE_{n+1} = x(t_{n+1}) - x_{n+1} \tag{3.7}$$

where $x(t_{n+1})$ is the exact solution to Eqn. (3.1) at t_{n+1}, and x_{n+1} is the computed solution obtained from Eqn. (3.2). In this definition, it is assumed that $x(t_n)=x_n$ and therefore it only provides information about the error which occurs over a single time-step, hence its name "local" truncation error. The LTE for the Forward-Euler method can be derived using Eqn. (3.4):

$$LTE_{n+1} = x(t_{n+1}) - x_n - h_n \dot{x}(t_n) \tag{3.8}$$

Using a Taylor expansion for the first term about t_n, the LTE is given by the first remainder term of the resulting expression:

$$LTE_{n+1} = \frac{h_n^2}{2}\ddot{x}(\xi) \qquad t_n \leqslant \xi \leqslant t_{n+1} \tag{3.9}$$

If E_A is some user allowable error tolerance for the problem, the accuracy constraint is:

$$\frac{h_n^2}{2}\ddot{x}(\xi) \leqslant E_A \qquad t_n \leqslant \xi \leqslant t_{n+1} \tag{3.10}$$

This presents a bound on the step size which is given by:

$$h_n \leqslant \sqrt{2E_A/\ddot{x}(\xi)} \tag{3.11}$$

If this constraint is not satisfied, the solution must be rejected and a new solution computed with a smaller step size. Since the exact value of ξ is not known, the LTE is

usually estimated using techniques to be described in a section to follow.

The Backward-Euler method has a LTE given by:

$$LTE_{n+1} = x(t_{n+1}) - x_n - h_n \dot{x}(t_{n+1}) \tag{3.12}$$

By expanding $x(t_n)$ in a Taylor series about t_{n+1} and applying the results to Eqn. (3.12), the LTE is obtained by retaining the first remainder term:

$$LTE_{n+1} = -\frac{h_n^2}{2}\ddot{x}(\xi) \qquad t_n \leqslant \xi \leqslant t_{n+1} \tag{3.13}$$

Note that the error made in one step is $O(h^2)$ in both the FE and BE methods, hence the accuracy bound on the step size is similar in both cases. However, the behavior of the global error, due to the accumulation of the local errors, may be quite different for the two methods and this difference strongly recommends the use of one method over the other. This characteristic is associated with the stability of the integration method.

b. Stability Constraint

The general stability characteristics of numerical integration methods applied to nonlinear differential equations are difficult to obtain. Usually the results are inferred from the analysis of a simple linear test problem [Gea71]:

$$\dot{x}(t) = -\lambda x(t) \ , \quad x(0)=x_0 \tag{3.14}$$

for which the solution is known to be

$$x(t) = x_0 e^{-\lambda t} \tag{3.15}$$

and, in general, λ is complex. This linear problem is useful because it is easy to analyze and provides information about the local behavior of nonlinear problems (i.e., when the step size is small). The problem is usually analyzed with $Re(\lambda)>0$ so that the solution to Eqn. (3.14) is stable. To further simplify the analysis, a fixed time-step is assumed. For example, if the FE method is used to solve Eqn. (3.14), the following difference equation is obtained:

$$x_{n+1} = x_n - \lambda h x_n = x_n - \sigma x_n$$

where $\sigma = \lambda h$. Therefore,

$$x_{n+1} = (1-\sigma)x_n = (1-\sigma)^{n+1} x_0$$

The region of Absolute Stability is defined as the set of all complex values of σ such that x_{n+1} remains bounded as $n \to \infty$. For FE, it consists of all σ such that

$$|1-\sigma| \leq 1 \tag{3.16}$$

which produces the following constraint for real values of λ:

$$0 \leq \sigma \leq 2.$$

Therefore the time-step must lie in the range:

$$0 \leq h \leq \frac{2}{\lambda}. \tag{3.17}$$

If step sizes outside this range are used, the computed solution will become unstable even though the true solution is stable. For BE, the difference equation is:

$$x_{n+1} = x_n - \sigma x_{n+1}$$

Hence:

$$x_{n+1} = \frac{1}{(1+\sigma)^{n+1}} x_0$$

which results in the following requirement for stability:

$$\frac{1}{|1+\sigma|} \leq 1 \tag{3.18}$$

Considering only real values of λ, the method produces a stable solution for all $h \geq 0$. Ideally, an integration method should produce a stable solution if the true solution is stable for any step size and this is the case for the BE method but not for FE. This property highly recommends the use of the BE method over the FE method since the step size can be selected based on accuracy considerations alone. For the general case when λ is complex, the region of Absolute stability for the BE integration method includes the entire right-half σ-plane. An integration method with this property is said to be A-stable [Dah63].

The Forward-Euler and Backward-Euler methods are examples of first-order integration methods. Higher-order methods with smaller local truncation errors can be constructed by taking more terms in the Taylor expansions of Eqns. (3.4) and (3.6). Integration methods with small LTE's are preferred as they allow larger time-steps to be used. For example, the trapezoidal method is a second-order integration method given by:

$$x_{n+1} = x_n + \frac{h_n}{2}(\dot{x}_{n+1} + \dot{x}_n) \tag{3.19}$$

and is quite popular as it is the most accurate A-stable method [Dah63]. The LTE for the trapezoidal method can be shown to be [ChLi75]:

$$LTE_{n+1} = -\frac{h_n^3}{12}\frac{d^3x}{dt^3}(\xi) \qquad t_n \leqslant \xi \leqslant t_{n+1} \tag{3.20}$$

Since the error is $O(h^3)$, it is often the case that a much larger step size can be used, compared to the BE method, for a given value of E_A.

c. Stiff-Stability Constraint

Another consideration in the choice of integration methods is the issue of stiffness. A stiff problem is one that exhibits time-scale variations of several orders of magnitude in the solution. A simple example of stiffness is the case of a fast initial "transient" in the solution, which dies quickly, followed by a slower "steady-state" solution. To handle this type of behavior, it is natural to use small time-steps in the transient portion to accurately follow the solution and then to increase the step size for the remainder of the solution. However, this strategy may lead to instability of the integration method, especially for explicit integration methods. For example, if the test problem in Eqn. (3.14) is solved using FE in the interval $[0;10^6\tau]$, where $\tau = 1/\lambda$, and $\lambda \in \mathbb{R}$, the time-step constraint given in Eqn. (3.17) would be imposed in the entire interval even though the solution decays to zero in approximately 5τ. If the step size is increased beyond this

stability bound, the solution will become unstable. On the other hand, if the size is kept within the constraint imposed by stability, the number of time points would be very large.

There are other situations which feature this kind of time-scale variation. A stiff problem is generated if the interval of time over which the system is integrated is large compared to the smallest time constant in the circuit, or if the circuit time constants themselves are widely separated. In addition, if the rise or fall time of an input waveform is widely separated from the circuit time constants, the problem also considered to be stiff.

Integration methods which are appropriate for solving stiff problems should have regions of Absolute Stability which cover most of the right-half complex σ-plane so that the time-step can be selected based on the accuracy considerations alone. Explicit methods are not well-suited to stiff problems since their regions of Absolute Stability are usually very small. The A-stable integration methods are well-suited to stiff problems, but other implicit methods (for example, see [ChLi75]) may be prone to instability when solving stiff problems. Gear proposed a family of integration methods called *stiffly-stable* methods [Gea71] which have the form:

$$\dot{x}_{n+1} = \frac{1}{h_n} \sum_{i=0}^{k} \alpha_i x_{n+1-i} \tag{3.21}$$

The values for α_i are chosen such that a k th-order method is exact if the true solution is a k th-order polynomial. The methods of order k=1 and k=2 are both A-stable algorithms. The methods of order k=3 up to k=6 are not A-stable, but they do have stability regions which are quite suitable for the integration of stiff problems [Gea71]. These methods are also referred to as Backward-Differentiation Formulas (BDF) [Bra72]. A variable-order method, also proposed by Gear [Gea71], uses the integration order which allows the largest step size at each time point. This technique was implemented in the

SPICE2 program [Nag75] and it was found that, even though the order could varied from k=1 up to k=6, a second-order method was used most often in the computation. The reason for this was attributed to the nature of the nonlinearities in the circuit simulation problem (described in the next section) and nature of the solution waveforms. Therefore, most circuit simulators use a low-order implicit integration method with guaranteed stability properties so that the step sizes can be selected based on accuracy considerations alone.

3.2.2. Solution of Nonlinear Equations

When solving linear dynamic circuits the accuracy and stability requirements of the numerical integration method are the only constraints on the step size used. Furthermore, linear problems can be solved in one "iteration" (i.e., one matrix solution) at each time point. Therefore, the amount of computation is directly proportional to the number of time points used. This is not true for nonlinear dynamic circuits, assuming that an implicit integration method is used. In fact, the cost of computing a solution at each time point is a function of the number of iterations used to solve the nonlinear algebraic problem. Consider the differential equation

$$\dot{x}(t) = f(x(t)) \tag{3.22}$$

where $f(x)$ is some nonlinear function. If the BE method is used to solve Eqn. (3.22), the following equation is obtained:

$$x_{n+1} = x_n + hf(x_{n+1}) = G(x_{n+1}) \tag{3.23}$$

This nonlinear algebraic equation can be solved using a variety of techniques including fixed-point iteration and Newton's method. The approach usually taken in circuit simulators is to use Newton's method or one of its variants. Rewriting Eqn. (3.23) as

$$F(x_{n+1}) = x_{n+1} - x_n - hf(x_{n+1}) = 0 \tag{3.24}$$

the Newton method to solve this equation is given by the expression [OrRh70]:

$$x_{n+1}^{k+1} = x_{n+1}^{k} - F(x_{n+1}^{k})/F'(x_{n+1}^{k}) \qquad (3.25)$$

where k is the Newton iteration counter. In circuit terms, the Newton method replaces each nonlinear device in the circuit by a linearized model based on operating point information. This process converts the nonlinear circuit into a linear equivalent network. The linearized network is solved using standard linear circuit analysis techniques [ChLi75]. The Newton method involves repeating the above steps until convergence is obtained.

To guarantee convergence of the Newton method, the function $F(x)$ and $F'(x)$ must be continuous in an open neighborhood about x^*, $F'(x^*) \neq 0$, and the initial guess, x^0, must be close enough to final solution. The Newton method is preferred over the simpler fixed-point method for several reasons. The main reason is that the fixed-point algorithm is not well-suited to stiff problems. It also imposes a bound on the time-step to guarantee convergence. Another reason is due to the quadratic convergence property of the Newton method. That is, if, in addition to the above conditions, $F''(x^*)$ exists, then for some $k > K$ the difference between successive iterations and the true solution satisfies the relation [OrRh70]:

$$|x^{k+1} - x^*| \leq c\,|x^k - x^*|^2 .$$

In practice, this quadratic convergence behavior occurs close to the final solution. Hence, it is important to provide an initial guess which is close to the final solution. In general, it is difficult to provide a reasonable starting guess for the Newton method. However, for the transient analysis problem it is possible to generate a good initial guess, especially if a capacitor exists between each node and the ground node[3]. For example, the solution at the previous time point is a good starting guess for the Newton method at t_{n+1}. A better approach is to use an explicit integration method [Bra72]:

[3] A capacitor to ground at each node implies some smoothness in the solution since it prevents instantaneous changes in the voltage at the node. Therefore, as $h \rightarrow 0$, $x_{n+1} \rightarrow x_n$.

$$x_{n+1}^{(0)} = \sum_{i=1}^{k+1} \gamma_i \, x_{n+1-i} \tag{3.26}$$

where the γ_i values are obtained by requiring that the predictor, $x_{n+1}^{(0)}$, be correct if the solution is a k th-order polynomial. Usually a k th-order predictor is used with a k th-order integration method.

The time-step also has some influence on the convergence speed of the Newton method. An intuitive reason for this can be given in circuit terms: the Newton method converts a nonlinear circuit into an associated linear circuit, as mentioned previously. As the step size is made smaller, the values of linearized circuit elements begin to approach their values at the previous time point. Therefore, the circuit will behave almost linearly in this interval and convergence can be obtained in very few iterations, possibly even a single iteration. On the other hand, if the step size is too large, a good starting guess may be difficult to generate, and this may lead to either slow convergence or nonconvergence. If nonconvergence should occur, the time-step must be rejected and a smaller step used in its place. Hence, in some cases, it may actually be more efficient to use two small steps rather than one large step.

3.3. TIME-STEP CONTROL IMPLEMENTATION ISSUES

The simplest time-step selection scheme is to use the same time-step throughout the interval of interest, $[0,T]$. That is, use a *fixed* time-step. Unfortunately, there are a number of constraints on the step size which may require that h be extremely small, resulting in a large number of time points. These constraints arise from the accuracy, stability and stiff-stability properties of a numerical integration method. For a fixed-step approach, the step size would have be chosen such that it satisfies these constraints under worst-case conditions. A better approach is to vary the step size during the simulation in accordance with the variation in the constraints. For a given problem, the allowable step sizes depend primarily on the properties of the specific integration

method being used. In this section, the main considerations in the implementation of an efficient time-step control for circuit simulation are described. It includes a discussion of LTE time-step control, iteration count time-step control and the effect of input sources on time step selection.

3.3.1. LTE Time-Step Control

In LTE time-step control, the user provides two accuracy control parameters, ϵ_a and ϵ_r, which are the absolute and relative errors permissible in each integration step. They are combined to form a user error tolerance:

$$E_{UserLTE} = \epsilon_a + \epsilon_r \times \max | x_{n+1}, x_n |$$

The general form of the local truncation error for most multistep integration methods of order k is given by [Gea71,ChLi75]:

$$LTE_{n+1} = \tilde{C}_k h^{k+1} x^{(k+1)}(\xi) \qquad t_n \leqslant \xi \leqslant t_{n+1} \tag{3.27}$$

where $\tilde{C}_k$ is a constant which depends on the coefficients of Eqn. (3.2) and the order of the method. Since the value of $x^{(k+1)}(\xi)$ is not known, in general, it must be estimated in some way using the numerical solutions. Typically a divided-difference approximation is used. The first divided-difference is defined as:

$$DD_1(t_{n+1}) = \frac{x_{n+1} - x_n}{h_n}$$

and the $k+1$st divided-difference is defined as:

$$DD_{k+1}(t_{n+1}) = \frac{DD_k(t_{n+1}) - DD_k(t_n)}{\sum_{i=0}^{k-1} h_{n-i}}$$

Then the estimate for the derivative term in Eqn. (3.27) is (see [Nag75] for derivation):

$$x^{(k+1)}(\xi) \approx (k+1)!\, DD_{k+1}(t_{n+1}).$$

The LTE estimate is then:

$$E_k = C_k h^{k+1} DD_{k+1}(t_{n+1})$$

For the BDF integration methods [Bra72], the LTE can be estimated in a more convenient way. The estimate is calculated using difference between the computed solution x_{n+1} and the predicted value $x^P(t_{n+1})$. For a k th-order BDF method, the following expression is used:

$$E_k = \left| \frac{h_n}{t_{n+1} - t_{n-k}} \right| (x_{n+1} - x^P(t_{n+1}))$$

The expression for $x^P(t_{n+1})$ is given in Eqn. (3.26). The computed solution x_{n+1} is accepted if

$$|E_k| < E_{UserLTE} \tag{3.28}$$

One way of implementing this check is to take the ratio of the allowable LTE and the actual LTE:

$$r = \frac{|E_{UserLTE}|}{|E_k|} = \frac{|C_k h_{allowable}^{k+1} \ddot{x}(\xi)|}{|C_k h_n^{k+1} \ddot{x}(\xi)|}$$

Noting that both errors are $O(h^{k+1})$, it follows that:

$$r = \left| \frac{h_{allowable}}{h_n} \right|^{k+1}$$

and

$$r_{LTE} = \frac{h_{allowable}}{h_n} = (r)^{\left(\frac{1}{k+1}\right)}$$

The comparison test given in Eqn. (3.28) becomes:

$$r_{LTE} > 1.0$$

to accept the computed solution. The advantage of this ratio is that it can also be used to select the step size for the next integration step. Therefore, the next recommended step size is given by:

$$h_{n+1} = r_{LTE} h_n \tag{3.29}$$

In practice, Eqn. (3.29) may occasionally recommend rather abrupt changes in the step size. A number of experiments have shown that rapid changes in step size may

introduce stability problems [Bra72]. Intuitively, the step sizes should follow the smoothness of the solution. To ensure that the changes in the step size are indeed gradual, it is convenient to set upper and lower bounds on the changes in step size. In fact, three regions can be defined as follows:

- if $r_{LTE} < 1.0$, reduce the step size by MAX(s_l , r_{LTE})
- if $1.0 \leq r_{LTE} < \alpha$, maintain the same step size
- if $r_{LTE} \geq \alpha$, increase the step size MIN(s_u , $\beta\, r_{LTE}$)

The time-step may be reduced by at most by the factor s_l and increased by at most by the factor s_u. The α factor permits the same step size to be used a number of times. Typically, α=1.2, s_l=0.25 and s_u=2.0. Note that a multiplying factor β has also been introduced as part of the growth factor. The β factor is a way of making the time-step selection somewhat conservative. Since the LTE can only be estimated, it may occasionally be optimistic [Yan80]. If so, the time-step would be rejected and a smaller step used unnecessarily. The β factor reduces the likelihood of this happening and a typical value is 0.9.

3.3.2. Iteration Count Time-Step Control

As mentioned before, the use of large steps is not necessarily the most efficient approach for nonlinear circuits, especially if relaxation is used. In fact, if time-step is too large, the iterative method may not converge, which would force the time-step to be rejected, resulting in wasted effort. This suggests that the time-step control should also be controlled by the nonlinearity of the problem.

A number of programs use a time-step control based on nonlinearity considerations alone (e.g., SPICE2, ADVICE, MOTIS3) called *iteration count* time-step control. This strategy minimizes the total number of Newton iterations used during the simulation. The step sizes are selected as follows. If the number of iterations is larger than N_{high}, the step size is reduced by some factor. If the number of iterations is less than

N_{low}, the step size is increased by some factor. Otherwise, the step size remains the same. The idea is to use approximately the same number of iterations at each time point.

While this strategy is certainly effective at reducing the overall computation time, it is prone to accuracy problems [Nag75]. For example, for linear circuits the step size would always be increased since the solution is always obtained in one "iteration" at each time point. For weakly nonlinear circuits, the same sort of effect would be observed. Therefore, this approach, when used by itself, is not recommended since it does not control the numerical integration errors directly. However, the iteration count time-step control can be used in conjunction with the LTE-based time-step control. In this case, if too many iterations were required to converge, a somewhat smaller step size could be used in the next integration step. If too few iterations are used a slightly larger step size can be used. The method could be implemented by making the growth factor dependent on the number of iterations used to compute the solution. Of course, if convergence is not obtained in a specified number of iterations, the time-step should be rejected and a smaller step used in its place.

3.4. LATENCY AND MULTIRATE BEHAVIOR

Most circuit simulators employing direct methods use a single common time-step for the whole system and hence compute the solution of every variable at every time point. The time-step at each point is based on the fastest changing variable in the system, i.e. the $n+1$st time point is given by:

$$t_{n+1} = t_n + h_n$$

where h_n is the integration step size determined by

$$h_n = \min(h_{1,n}, h_{2,n}, \cdots, h_{N,n})$$

and $h_{i,n}$ is the recommended step size for with the ith variable at t_n. As a result, many variables are solved using time-steps which are much smaller than necessary to compute their solution accurately. For example, the computed points of a waveform from a large digital circuit, simulated using direct methods, are shown in Fig. 3.1(a). Note that there are many more points than necessary to represent the waveform accurately, especially in the regions when the waveform is not changing at all. The extra points are due to some other variable changing rapidly in the same region of time. The same waveform is shown Fig. 3.1(b) with only the minimum number of points necessary to represent it accurately.

Since the objective in circuit simulation is to provide an accurate solution while minimizing the number of expensive model evaluations, one way to achieve this goal is to reduce the number of time points computed for each waveform. A number of circuit simulators have attempted to improve the efficiency in this manner by exploiting a property of waveforms called *latency* [Nag75, New78, Rab79, Yan80, Sak81]. While the general concept of latency includes any situation where the value of a variable at a particular time point can be computed accurately using some explicit formula, it usually refers to the situation where a variable is not changing in time and its solution can be obtained from the explicit equation:

$$x_{n+1}=x_n \tag{3.30}$$

That is, the value x_{n+1} is not computed using a numerical integration formula but instead is simply updated using the value at the previous time point. For example, the waveform shown in Fig. 3.2(a) has three latent periods, and ideally the value of x does not need to be computed in any of these regions.

In the SPICE program [Nag75], latency exploitation is performed using a *bypass* scheme. In this technique, each device is checked to see if any of its associated currents

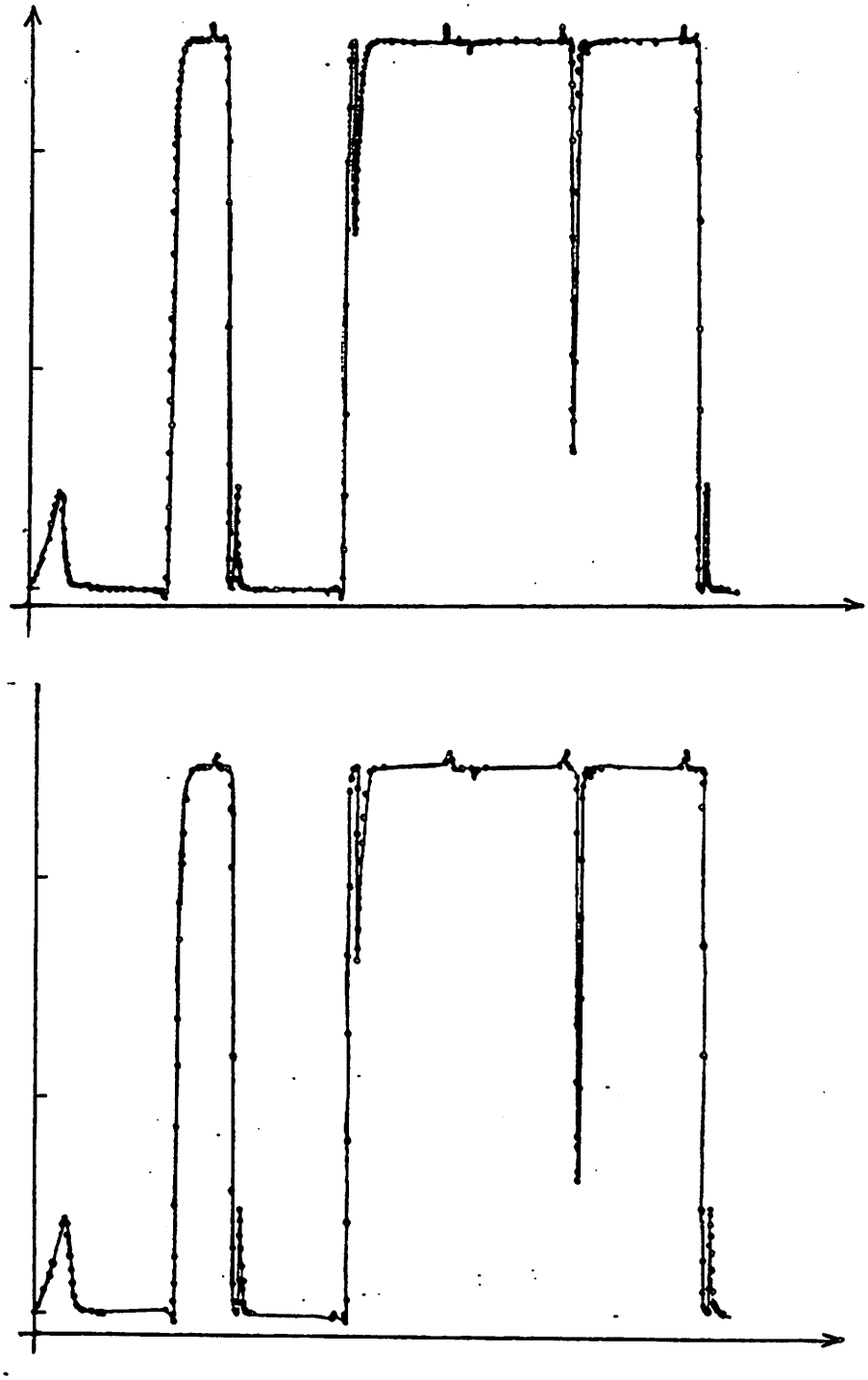


Figure 3.1 : Effect of solution by direct methods

and node voltages have changed significantly since the last iteration. If not, the same device conductances and current are also used in the next iteration. However, the checking operation is somewhat expensive, especially if the circuit is large and most of the devices are latent. In general, latency exploitation involves the use of a model describing the behavior of a particular variable as a function of time over a given interval. The simple model described in Eqn. (3.30) can be considered as a "zeroth-order" latency model. Higher-order latency models can be constructed if the solution is known to have a specific form (i.e., polynomial, exponential) or if the solution for the variable can be obtained in closed form. For example, a first-order latency model given by:

$$x_{n+1}=x_n+h_n\frac{I}{C}$$

can be used in the case of an ideal current source, with current I, charging a linear capacitor, C. Usually a latency model can only be used over a portion of the simulation interval. Therefore, the validity of the model must be monitored and its use must be discontinued when the model is thought to be invalid. The latency model used in this context has also been called a *dormant* model [Sak81].

In practice, only the zeroth-order form of latency can be exploited easily since the higher-order forms are difficult to construct for general nonlinear circuits. To exploit this simple form of latency, some mechanism is necessary to detect that the signal value is not changing appreciably[4]. The waveform is considered to be latent at that point, and its associated variable is updated using Eqn. (3.30) at subsequent time points. A second mechanism is used to determine when the latency model is invalid, and from that point onward the variable is computed in the usual way. Hence, the waveform is only computed at time points when the signal is changing. Event-driven, selective-trace can be

[4] A number of schemes to detect zeroth-order latency are described in Chapter 4.

used to exploit latency, as described in the next chapter, without incurring the overhead of the bypass scheme.

It is only useful to exploit this simple form of latency when some variables in the circuit are changing while other variables are stationary, since direct methods can adequately handle the case when all variables are active or latent. In fact, the "useful" form of zeroth-order latency can be viewed as a subset of a more general property of waveforms called *multirate behavior* which is illustrated in Fig. 3.2(b). Multirate behavior refers to signals changing at different rates, relative to one another, over a given interval of time. MOS circuits inherently exhibit this kind of behavior because of different transistor sizes and different capacitance values at each node. Exploiting this general property can reduce significantly the number of time points computed for each waveform since large steps can be used for variables changing very slowly while smaller steps can be used for rapidly changing variables.

The basic strategy to speed up circuit simulators suggested above is to take advantage of the relative inactivity of large circuits by reducing the number of time points computed. However, the actual speed improvement obtained by solving the equations in this manner depends on the two main factors:

1) the "amount" of latency and multirate behavior exhibited by the circuit during the simulation, and

2) the efficiency of techniques used to exploit the two properties.

The first point refers to the maximum speed improvement that can be obtained if the two waveform properties are exploited fully, and this factor depends on the circuit size and the activity in the circuit generated by the external inputs. The second factor depends on the actual number of points computed and the work required to compute each point. In the remainder of this section, simple experiments are given to compute the speed improvement that can be obtained by exploiting latency and multirate

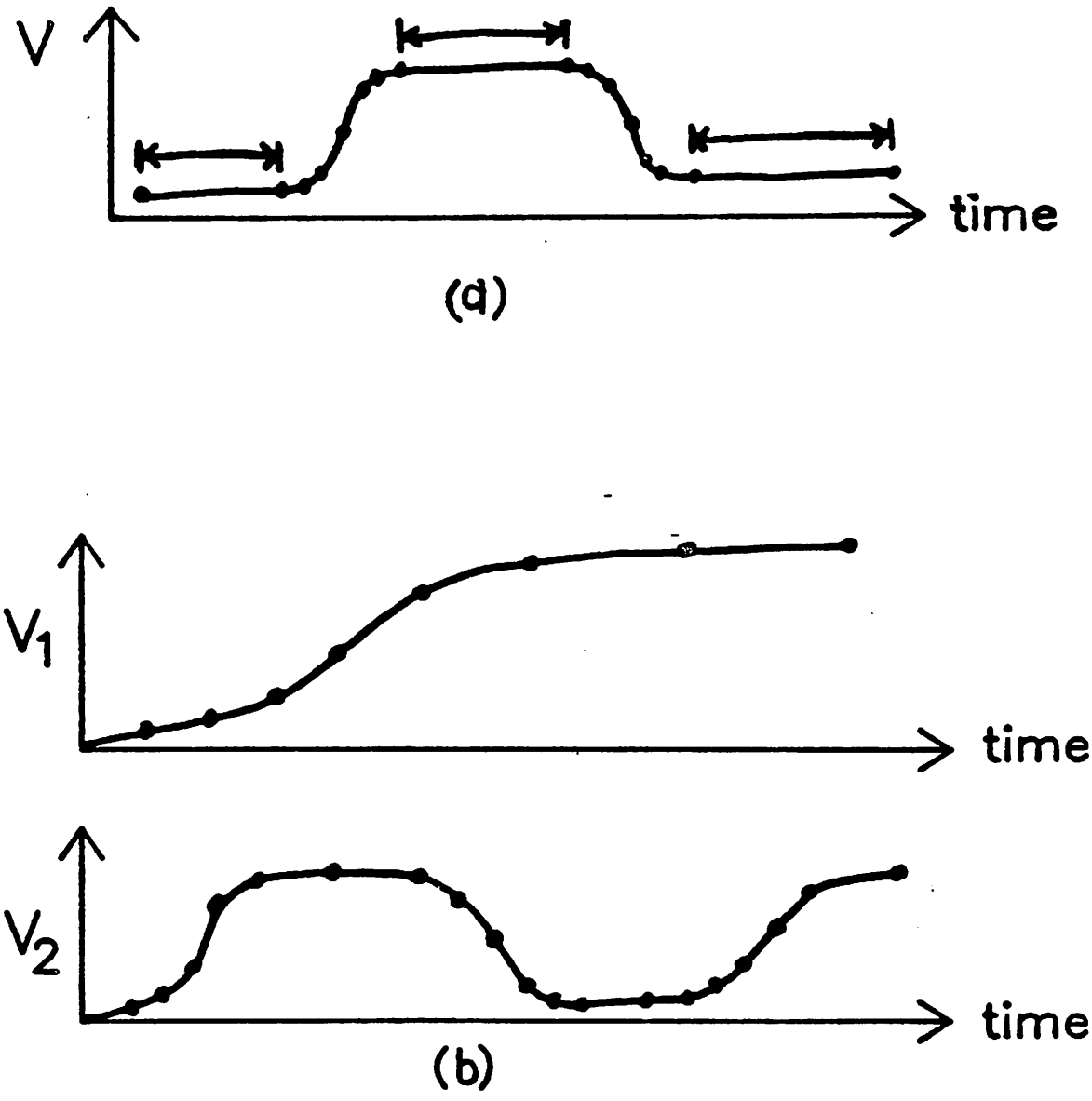


Figure 3.2: Waveform Latency and Multirate Behavior

behavior under ideal conditions. The efficiency issue is addressed in the Section 3.5. One assumption used in the following is that work required to generate a solution at each time point is constant and independent of the method used to produce the solution. That is, the number of model evaluations per node per time point is assumed to be fixed. The validity of this assumption will be checked in Section 3.5 since it is related to the efficiency issue.

3.4.1. Maximum Speed-up if Latency is Exploited

Assume that a circuit is simulated using direct methods and that an accurate solution is computed at T_{LTE} time points. Note that this is the number of points at which the LTE criterion was satisfied. In reality, a few solutions may have been rejected due to both LTE rejections and iteration count rejections. The total number of computed time points, T_D, is given by:

$$T_D = T_{LTE} + T_{LTErej} + T_{Nrej}$$

where T_{LTErej} is the number rejected due to LTE rejections and T_{Nrej} is the number rejected due to iteration count rejections. Furthermore, since a conservative time-step control is used in practice, the actual step sizes used are smaller than the step sizes allowed by the LTE. That is, $T_{\dot{L}TE} \leq T_{LTE}$, where $T_{\dot{L}TE}$ is defined as the number of points that would be used if the maximum step size allowed by the LTE criteria is used, without any conservatism.

If there are N nodes in the circuit then, assuming that the CPU-time is proportional to the number of points computed, the amount of work required to simulate the circuit using direct methods is

$$W_D = N \times T_D .$$

Now the question is: by how much can the speed be improved if waveform latency is exploited, assuming ideal conditions? Two conditions would make the situation ideal.

First, if a node-by-node decomposition of the circuit is used, it would allow each equation to be solved independently so that latency can be exploited for each waveform. Second, if the latency condition is identified without error, as soon as it occurs in the waveform, the fewest number of points would be computed.

In order to compute an upper bound on the speed improvement, the waveforms from a simulator using direct methods are necessary. These waveforms can be obtained by performing a simulation using direct methods and saving the output waveforms for all nodes. An operation can be performed on each waveform to determine how many points would be computed if latency could be exploited fully. A simple post-processor is given in Algorithm 3.1 to perform this operation on each waveform.

Algorithm 3.1 (Zeroth-Order Latency Evaluator)

```
    /* y(t) is the given input waveform */
    /* x(t) is the output waveform */
    k ← 0 ; n ← 0 ;
    x(t_0) ← y(t_0);
    while (t_k ≤ T_stop) {
        k ← k +1 ;
        while ( |x(t_k) − y(t_n)| < threshold )
            n ← n +1 ;
        x(t_k) ← y(t_n) ;
    }
    output x(t );
```

■

In words, each computed point in the waveform $y(t)$ is categorized as being an active point or a latent point using some threshold of significant change between time points. Only the active points are moved from waveform $y(t)$ to waveform $x(t)$ and provided as output by the post-processor. For the purposes of this post-processor, the use of an absolute threshold to detect latency is permissible, as long as it is chosen properly, since the only objective is to remove unnecessary points from each waveform. However, the proper detection of latency during a simulation is an important issue which is described in detail in Chapter 4.

If T_{Li} is the number of latent time points removed from waveform i, the available speed improvement if latency is exploited is given by

$$S_L = \frac{W_D}{\sum_{i=1}^{N}(T_D - T_{Li})}. \tag{3.31}$$

This value can be considered as an upper bound on the speed improvement since node-by-node decomposition is used and the latency detection is assumed to be ideal.

3.4.2. Maximum Speed-up if Multirate Behavior is Exploited

The next step is to compute an upper bound on the speed improvement if multirate behavior is exploited. In this case, the idea is to determine the minimum number of points required to represent each waveform in the circuit accurately, i.e., given the waveform in Fig. 3.1(a), produce the equivalent waveform in Fig. 3.1(b). A node-by-node decomposition is also assumed here. As before, the waveform of Fig. 3.1(a) can be obtained from a simulator using direct methods. The waveform corresponding to Fig. 3.1(b) is generated by the multirate post-processor, as described in the following.

The post-processor selects an integration method and an accuracy criteria, E_A, and replaces the set of points in a given waveform with a new set of points that only depends on the characteristics of the waveform itself. That is, the only criteria used to select step sizes is the LTE. As described earlier, different integration methods have different local truncation errors. Usually higher-order integration methods allow larger step sizes resulting in fewer computed points. The allowable error also controls the number of time points used. For example, if E_A is very small, the number of points generated by the post-processor will be very large. Therefore, the order of integration and the accuracy criteria should be the same ones used in the simulator using direct methods to get meaningful results from this analysis.

Another consideration is that, in practice, the LTE is not the only constraint on the step size. The iteration count also controls the selection of step sizes. Since the post-processor computes $T^{*}_{LTE,i}$ points for waveform i, and it is known that

$$T^{*}_{LTE,i} \leq T_{LTE,i} + T_{Nrej,i},$$

it follows that the post-processor will produce the minimum number of points possible, thus providing a more conservative upper bound.

In the algorithm below, the waveform, $y(t)$, is the input to the post-processor and is treated as the correct solution. The post-processor attempts to choose optimal step sizes which satisfy the LTE constraint. It uses the BDF method [Bra72] to compute the LTE. That is, if the difference between the predicted value and the computed solution is within the specified tolerance, the step size is accepted. Otherwise a smaller step is used and the process is repeated. Since the trapezoidal method is a popular integration method for circuit simulation due to it accuracy and stability properties, a second-order integration method is assumed in the evaluator. The sequence of operations in the post-processor is given more precisely in Algorithm 3.2.

Once this post-processing operation is completed, the number of points, $T^{*}_{LTE,i}$ used for each waveform are tabulated and compared it to the number of points generated by the simulator using direct methods. Then, the upper bound on the speed improvement is computed as

$$S_M = \frac{W_D}{\sum_{i=1}^{N} T^{*}_{LTE,i}} \tag{3.32}$$

If $S_M = 1$, the circuit exhibits no multirate behavior. However, if $S_M >> 1$ then the circuit is said to be "highly multirate".

Algorithm 3.2 (Multirate Evaluator for 2nd-Order Integration Method)

```
/* y(t) is the given input waveform */
/* x(t) is the output waveform */
k ← 0 ;
t_k ← 0 ;
x(t_0) ← y(t_0);
while ( t_k ≤ T_stop ) {
    k ← k +1 ;
    pick a new step size h_k ;
    stepAccepted = FALSE ;
    repeat {
        t_k ← t_k-1 + h_k ;

        Get predicted value x^P(t_k)
          using 2nd-Order predictor using x(t_k-1), x(t_k-2), x(t_k-3) ;

        Get computed value y(t_k)
          using 2nd-order interpolation of the waveform y(t) at t_k ;

        E_2 = [h_n /(h_n + h_n-1)](y(t_n+1) - x^P(t_n+1)) ;
        E_A = ε_a + ε_r × max |x^P(t_k), y(t_k)| ;
        r_LTE = (E_A / E_2)^1/3;

        if (r_LTE < 1.0) h_k ← h_k /2.0;
        else if ( 1.0 ≤ r_LTE ≤ 2.0 ) stepAccepted = TRUE ;
        else if (r_LTE > 2.0) h_k ← h_k + h_min

    } until ( stepAccepted );
    x(t_k) ← y(t_k) ;
}
output x(t );
```

■

3.4.3. Experimental Results

A number of industrial circuits were simulated and processed as described above and the results are presented in Table 3.1. The SPLICE3 simulator has an implementation of direct methods[5] and this mode was used to generate the results. A wide variety of circuits are represented in this table including an opamp, a control circuit, a few memory circuits and a digital filter circuit. The last two columns in the table show the

[5] The SPLICE3 program is described in Chapter 4 and Chapter 5. The implementation of direct methods in this program is based on the direct methods used in the RELAX2 program. However, the time-step control is slightly different in SPLICE3 since it allows latency exploitation.

upper bounds on the speed improvements if latency and multirate behavior are exploited fully using Eqns. (3.31) and (3.32).

As the results indicate, there is a strong incentive to take advantage of the relative inactivity of large circuits. The potential speed-up factors range from 1.8 for the smallest circuit to almost 64 for the largest circuit. Note that the factors may vary somewhat depending on the circuit and the length of the simulation interval. In fact, even for two circuits of roughly the same size (i.e., CRAMB and SCDAC), the speed-up factors are different. However, the general trend is that the potential speed-up increases as the circuit size increases. Note also that the bounds given in Table 3.1 are only meaningful when the model evaluation time is the dominant factor in the total time, which is the case for small and medium size circuits. If the linear equation solution time is the dominant factor for direct methods, then the actual speed improvement could exceed this upper bound.

Circuit	Size (nodes)	Available Speed-up (Latency)	Available Speed-up (Multirate)
OPAMP	13	1.8	2.3
DECPLA	56	2.2	4.6
CRAMB	149	12.3	27.9
SCDAC	154	4.1	8.5
CKT3	312	6.5	25.9
DIGFI	378	8.2	16.8
EPROM	630	18.4	63.9

Table 3.1 - Upper bounds on speed improvement for some industrial circuits if latency and multirate behavior are exploited fully.

3.5. EFFICIENCY OF RELAXATION METHODS

The results in Table 3.1 indicate that it is worthwhile to explore techniques which take advantage of the latency and multirate properties of waveforms. To accomplish this, the equations must solved in a decoupled fashion so that different time-steps may be used to solve different equations. Clearly the relaxation methods described in Chapter 2 are well-suited for this purpose. A variety of relaxation algorithms are described in the chapters to follow and they are shown to be effective in exploiting both latency and multirate behavior. However, one may wonder if the bounds given in Table 3.1 are achievable in practice, and what factors control efficiency of the relaxation methods in this respect.

The overall efficiency of a method is related to the number of points computed per node and the cost of each computed solution. If these two factors are examined for relaxation methods, it is apparent that the bounds in Table 3.1 will be difficult to reach. For example, the factors which *increase* the number of computed points are as follows:

- the time-step selection is usually conservative in most circuit simulators to reduce the likelihood of a step rejection and this increases the number of points computed.

- in the multirate post-processor, the ideal time-steps were chosen based on a LTE criteria alone. However, the actual step size will be smaller whenever the iterative process does not converge, thereby increasing the number of computed points.

- in the latency post-processor, the latency detection scheme was assumed to be ideal. In reality, the techniques used to detect latency must be very conservative so that errors are not made. This also increases the number of points computed.

- the node-by-node decomposition assumed throughout the discussion allows the maximum amount of latency and multirate behavior to be exploited but is not appropriate for general circuits. The use of circuit partitioning increases the total number of points computed, since variables in the same subcircuit are solved together. However, this usually improves the convergence speed enough to warrant its use.

Based on the above considerations, it is clear that the number of time points computed

in practice will be larger than the number used to compute the bounds.

The second factor is the relative cost of computing a solution at each time point compared to direct methods. It was assumed earlier that the number of model evaluations per node per time point is the same regardless of the method used to compute the solution. Strictly speaking, this assumption is not even true for direct methods on nonlinear problems. The cost depends on the number of Newton iterations at each time point, which in turn depends on the step size and the nonlinearity of the problem in the neighborhood of the time point. However, the time-step control algorithm suggested in Section 3.3 attempts to make the cost approximately equal at all time points by adjusting the time-step based on iteration count. For the purposes of this analysis, it is assumed that the cost of each solution point is constant for direct methods.

In the remainder of this section, the cost of computing a solution point for the relaxation methods is examined. To simplify the analysis, consider the case where all methods use the same fixed step size to perform a simulation in the interval $[0,T]$. For direct methods, if there are d devices in the circuit, then the number of model evaluations performed at each time point is $\bar{N}_R d$, where $\bar{N}_R$ is the average number of Newton iterations. For nonlinear relaxation, each subcircuit is processed once during each relaxation iteration. If $\bar{R}_{NL}$ is the average number of nonlinear relaxation iterations at each time point, then the number of model evaluations is given by $\frac{\bar{R}_{NL} d}{\alpha}$, where $\alpha > 1$. Note that α accounts for the fact that all entries in the Jacobian matrix will not be computed in this case for reasons described in Chapter 2. Therefore, the device evaluations will require less computation and α is used to reflect this fact. For WR, each subcircuit requires an average of $\bar{R}_{WR}$ relaxation iterations to converge to the solution. During each relaxation iteration, an average of $\bar{N}_R$ Newton iterations are used to solve the nonlinear problem at each time point. Therefore, the total number of model evaluations

per time point is given by $\frac{\bar{N}_R \bar{R}_{WR} d}{\alpha}$.

By comparing the three expressions for the number of model evaluations per time point, it is evident that the cost can be very high for WR if $\bar{R}_{WR}$ is large. However, this higher cost is offset by the fact that fewer time points are computed in WR since it exploits the multirate property. The cost per solution point for direct methods and nonlinear relaxation are approximately the same if $\bar{N}_R \approx \bar{R}_{NR}/\alpha$. For example, if half as much computation is performed during the model evaluation phase in nonlinear relaxation (i.e., α=2) and $\bar{R}_{NR} = 2\times\bar{N}_R$, then the cost per time point would be the same as direct methods. This suggests that if nonlinear relaxation is used to exploit waveform latency, the speed-up numbers in the latency column of Table 3.1 will only be modified by the actual number of time points computed compared to the ideal number, and not by the cost of each solution point. For WR, the multirate speed-up numbers in the table will be affected by *both* the number of relaxation iterations and the actual number of time points compared to the ideal value, T^*_{LTE}, used in the upper bound calculations.

CHAPTER 4
ITERATED TIMING ANALYSIS

4.1. INTRODUCTION

In this chapter, a number of algorithms for the solution of the circuit simulation problem based on nonlinear relaxation are described. It is shown that nonlinear relaxation combined with *event-driven selective-trace* techniques can be used to exploit latency or circuit inactivity. This approach is referred to as Iterated Timing Analysis or ITA [Sal83]. A preliminary version of ITA was implemented in the prototype mixed-mode simulator SPLICE1.7 [Sal84] and an advanced version in SPLICE2 [Kle84]. A new robust version of ITA has been implemented in the SPLICE3.1 program. The details of the implementation of ITA in these programs are given in this chapter.

In Section 4.2, nonlinear relaxation is applied to the circuit simulation problem. Simple timing analysis is described in Section 4.3. Fixed step ITA is introduced in Section 4.4, and global-variable time-step ITA is described in Section 4.5. A number of issues concerning latency and event scheduling are presented in Section 4.6. Simulation results are reported in Section 4.7 and conclusions are given in Section 4.8.

4.2. EQUATION FLOW FOR NONLINEAR RELAXATION

The starting point for the description is the system of nonlinear differential equations describing the circuit behavior using the charge-based formulation:

$$\dot{q}(v(t)) = -f(v(t),u(t)), \quad v(0)=V, \quad t \in [0,T] \tag{4.1}$$

where q is the charge associated with the capacitors connected to each node, f is the sum of the currents charging the capacitances at each node, u is set of input voltages and v is the set of unknown node voltages. Using trapezoidal integration [ChLi75] to discretize the system in Eqn. (4.1), the following system of nonlinear difference equa-

tions is obtained:

$$q_{n+1} = q_n + \frac{h_n}{2}(f_{n+1} + f_n) \tag{4.2}$$

where the subscripts n and $n+1$ refer to time points t_n and $t_{n+1}=t_n+h_n$, respectively, and h_n is the integration step size. This equation can be formulated as a nonlinear problem, as follows:

$$F(v) = \frac{2}{h_n}(q_{n+1} - q_n) - (f_{n+1} + f_n) = 0 \tag{4.3}$$

Instead of solving this system of equations using standard techniques [Nag75], the strategy in this section is to use nonlinear relaxation. That is, use the Newton method to solve each equation in the system separately and a relaxation method to guarantee that the solutions are mutually consistent. The expression for the i th equation in Eqn. (4.3) solved using the Newton method is:

$$J_{F_i}(v^k)(v_i^{k+1} - v_i^k) = -F_i(v^k) \tag{4.4}$$

where $J_{F_i}(v)$ is the i th diagonal term of the Jacobian matrix of $F(v)$ given by:

$$J_{F_i} = \frac{2}{h}\frac{\partial q_i(v^k)}{\partial v} - \frac{\partial F_i(v^k)}{\partial v} \tag{4.5}$$

The index k is the iteration counter for the Newton method. Usually a number of iterations are required to obtain the correct solution. However, in this case, since a converged relaxation method is used to guarantee a consistent solution to the system of equations, the Newton iteration for each equation need not be carried to convergence. In fact, from an efficiency standpoint, *only one iteration* should be used to approximate the solution of each equation before moving to the next equation, as described earlier in Chapter 2. The resulting one-step Gauss-Seidel-Newton relaxation algorithm is specified precisely in the following, using the definition:

$$v^{k,i} = [v_1^{k+1}, v_2^{k+1}, \cdots, v_{i-1}^{k+1}, v_i^k, v_{i+1}^k, \cdots, v_n^k]^T$$

where the superscript T denotes the transpose of a vector. This definition is based on

the Gauss-Seidel method which uses the $k+1^{st}$ values of all other components, whenever possible, in computing the $k+1^{st}$ value of v_i. Here n is the number of equations in the system.

Algorithm 4.1 : (Gauss-Seidel-Newton Relaxation Method)

repeat {
  foreach ($i \in \{1, \cdots, n\}$) {
    solve $J_{F_i}(v^{k,i})(v_i^{k+1} - v_i^k) = -F_i(v^{k,i})$ for v_i^{k+1}
      where $F_i(v)$ is specified in Eqn. (4.3) and
        $J_{F_i}(v)$ is specified in Eqn. (4.5) ;
  }
} **until** ($\|v_i^{k+1} - v_i^k\| < \epsilon_1$, $\|F_i\| < \epsilon_2$, $i = 1, \ldots, n$)

■

4.3. TIMING ANALYSIS ALGORITHMS

The first published program to use techniques based on nonlinear relaxation for circuit simulation was the MOTIS program [Cha75]. It used Backward-Euler integration, a Gauss-Jacobi-Newton relaxation algorithm, and node-by-node decomposition (that is, it solved for one node voltage at a time). In MOTIS, a simple modification was made to the relaxation scheme based on the conjecture that there exists a small enough time-step, h_{min}, such that the method obtains the correct solution in exactly one iteration. At each time point, t_{n+1}, the program computed new values of all node voltages using only one iteration of the Gauss-Jacobi-Newton method and accepted the results as the correct solutions at t_{n+1}. It was believed that iterating the outer relaxation loop to convergence would be both expensive and unnecessary for most MOS logic circuits. However, the resulting accuracy of this approach relied heavily on three things:

(1) the user's ability to select an appropriate time-step based on knowledge of the circuit characteristics

(2) the fact that the global error reduces to zero when a node voltage reaches the supply voltage or ground

(3) Only a limited number of well-characterized circuit topologies (CMOS polycells) were used to build a design

The initial speed improvements obtained using this approach were extremely encouraging, partially due to the simplified numerical techniques and partially due to the use of table look-up models for the MOS devices. The combined techniques were shown to be over two orders of magnitude faster than standard techniques when applied to large digital MOS circuits [Cha75]. Since the method was intended to provide first-order timing information of MOS logic circuits, it was called "Timing Analysis" or "Timing Simulation".

Although timing analysis provided an electrical simulation capability with execution speeds comparable to logic simulation, it suffered from a number of problems. For example, the choice of a proper time-step to guarantee accurate solutions was very difficult to determine in general. In addition, the method had severe accuracy problems for circuits containing elements such as large floating capacitors[1], small floating resistors and transfer gates. The MOTIS program avoided this problem for floating capacitors by not allowing them in the circuit description and solved collections of transfer gates using direct methods.

A number of improvements to the basic technique were suggested to overcome the inherent accuracy limitations of the method. In particular, the MOTIS-C program [Fan77] employed trapezoidal integration and one iteration of the Gauss-Seidel-Newton relaxation algorithm. Since timing analysis algorithms based on the Gauss-Seidel principle use updated information at t_{n+1} whenever possible, the accuracy is generally better than one based on the Gauss-Jacobi method. The simulation time-step was selected automatically in the program by doing a simple analysis of the time constants associated with each node and used some fraction of the smallest time constant as the step size. However, MOTIS-C still suffered from problems similar to MOTIS.

[1] A "floating" element is a two-terminal device whose terminals are not connected to either ground or to a power supply.

A modified timing analysis algorithm was implemented in SPLICE1.3 [New78] as part of a mixed-mode simulation capability. Although Backward-Euler integration was used in this program, a number of other noteworthy enhancements were made to the underlying timing analysis algorithm. The first enhancement was based on two observations:

(1) most of the node voltages in a large digital circuit remain stationary at a given time point (the latency property). Computing the solution for these nodes is unnecessary.

(2) the order in which the nodes are solved has a strong influence on the accuracy of the solution for timing analysis algorithms based on the Gauss-Seidel principle

These observations suggested that a good strategy would be to identify the "active" nodes at each time point and process these nodes an order based on the direction of signal flow. In SPLICE1.3, a single mechanism was used to perform both tasks: an *event-drive*, *selective-trace* algorithm normally associated with logic simulation [SzTh75]. This mechanism is described in the following paragraphs.

The SPLICE1 program treats a circuit as a signal-flow graph and constructs a corresponding directed-graph for the circuit given by, $G = G(X, E)$, where X is the set of vertices and E is the set of directed edges of the graph. Two tables, the *fanin* and *fanout* tables, are constructed at each vertex based on the following definitions:

Definition 4.1: (Fanin and Fanout nodes)

A node x_k is called a fanin node of x_i, and is specified as $x_k \in Fanin(x_i)$, if x_k directly affects x_i. A node x_j is called a fanout node of x_i, and is specified as $x_j \in Fanout(x_i)$, if x_j is directly affected by x_i.

Whenever the value of an input node or any internal node changes, it is possible to *schedule* all of its fanouts to be processed. In this way the effect of a change at the

input to a circuit may be *traced* as it propagates to other circuit nodes via the fanout tables. Since the only nodes that are processed are those which are affected directly by the change, this technique is *selective* and hence its name: *selective trace*. If such a selective trace algorithm is used with the fanout tables, the order in which the nodes are updated becomes a function of the signals flowing in the network and is therefore a dynamic ordering.

To make the processing efficient, and for consistency with the logic simulator in the SPLICE1 program, the total simulation period, T_{stop}, is divided into uniform steps, referred to as the *Minimum Resolvable Time* (*mrt*). A time queue is constructed and the time slots in this queue define distinct points in time separated by one *mrt*. Hence, events are scheduled at integer multiples of *mrt* in the queue. The simple event scheduling algorithm used in SPLICE1 for timing analysis is given below. The routine *NextEventTime(t)* examines successive time slots in the time queue starting at time t and returns the next time point where one or more events have been scheduled. The external input nodes to a circuit are denoted as e_k.

Algorithm 4.2 : (Event Scheduling Algorithm in SPLICE1)

```
t_n ← 0;
while ( t_n ≤ T_stop ) {
    t_n ← NextEventTime( t_n );
    foreach ( input k at t_n ) {
        if ( e_k is "active" )
            forall ( x_j ∈ Fanout (e_k ) ) Schedule( x_j , t_n );
    }
    foreach ( event i at t_n ) {
        process node x_i by computing x_i (t_n );
        if ( x_i is "active" ) {
            Schedule( x_i , t_n + h );
            forall ( x_j ∈ Fanout (x_i ) ) Schedule( x_j , t_n );
        }
    }
}
```

■

As seen in the above, three separate event scheduling mechanisms exist:

(1) external inputs generate events whenever they make transitions from one value to another.
(2) internal nodes can schedule themselves to be processed, and
(3) internal nodes can schedule their fanout nodes to be processed.

Note that if x_i is not active, then neither x_i nor its fanouts are scheduled. However, since nodes may schedule themselves, the fanouts of x_i may still be active even though x_i is not. The importance of this fact and other issues associated with electrical event scheduling will be presented in Section 4.6. Also, the precise meaning of "active" is elaborated further in Section 4.6.

The use of event-driven, selective-trace techniques greatly improved accuracy of SPLICE1.3 compared to the MOTIS and MOTIS-C programs. In addition, a further improvement was realized using a variable time-step control, as follows. Initially, every node is solved using a common step size given by the *mrt* . If the change in either the voltage at a node or the current through any device connected to the node is large, its solution is recomputed in the *mrt* interval using smaller steps and a single iteration at each time point. Each of the smaller steps may be further refined to insure that the changes in voltage and current are within acceptable limits. Therefore, the local time-steps for each node are based on limiting change of the node voltage and its associated currents over each step[2]. While the run time was noticeably higher, this variable time-step control was extremely effective in improving the accuracy of the results.

Other enhancements were developed in SPLICE1.3 to handle tightly-coupled circuits. SPLICE1.3 used the Implicit-Implicit-Explicit (IIE) method [New80] to handle floating capacitors and this approach has been shown to be convergent, consistent and stable [Hua83]. To accommodate large blocks of tightly-coupled circuit elements, the

[2] Note that a variable time-step control based on local truncation error is not easy to define here since the relaxation loop is not carried to convergence. The local error (i.e., the error over one step) is due to the integration method and due to the fact that the iteration is not carried to convergence.

program allowed the user to define "circuit" blocks. These blocks would be solved using standard direct matrix techniques. However, instead of using a single iteration, the Newton iteration in the inner loop was carried to convergence since the elements inside the circuit block were considered to be "highly" nonlinear. However, the outer relaxation iteration was still only performed once.

While the results from programs using timing analysis were within acceptable accuracy limits for a certain class of problems, a rigorous mathematical analysis indicated that these methods have inherent stability and accuracy problems [DeM81]. This severely limited the application of the technique. Another problem, cited earlier, was that timing analysis programs relied on the user's knowledge of the underlying algorithms and improper usage could produce the wrong answer. Circuit designers have been known to lose confidence in a simulator if it occasionally produces the wrong answer, whatever be the reason. Therefore, this approach has not been widely accepted, although it is heavily used where the approach has been thoroughly developed, is well-understood, and is applied to a restricted class of circuit topologies.

4.4. SPLICE1.7 - FIXED TIME-STEP ITA

The reluctance to close the outer relaxation loop in timing analysis was primarily due to its perceived high cost. However, the event-driven techniques significantly reduced the cost of timing analysis for large problems since only a small fraction of the nodes are processed at each time point. A number of other improved timing analysis algorithms were proposed [Kah75, DeM83] but they used at least two iterations or required the use of expensive function evaluations, which increased greatly the cost of the simulation. As described earlier, the variable step approach in SPLICE1.3 improved the accuracy somewhat at the expense of additional iterations. The additional cost was thought to be worthwhile due to the improved reliability.

The next step, naturally, is to close the relaxation loop and examine the true cost of iterating to convergence, given that event-driven selective-trace is employed to improve efficiency. This was done in the SPLICE1.6 program, which later evolved to be SPLICE1.7, and the technique was named Iterated Timing Analysis or ITA [Sal82]. The prototype version of ITA used Backward-Euler integration, node-by-node decomposition and a fixed time-step based on the *mrt*. The fixed time-step algorithm was kept for consistency with the existing scheduler and logic simulation portions of SPLICE1. The ITA algorithm in SPLICE1.7 is a simple extension of Algorithm 4.3 as shown below.

Algorithm 4.3 : (Fixed Time-Step ITA)

```
t_n ← 0;
while (t_n ≤ T_stop) {
    t_n ← NextEventTime( t_n );
    foreach ( input k at t_n ) {
        if ( e_k is active )
            forall ( v_j ∈ Fanout (e_i) ) Schedule( v_j , t_n );
    }
    repeat {
        foreach ( event i at t ) {
            solve J_Fi(v^k,i)(v_i^k+1 − v_i^k) = − F_i(v^k,i) for v_i^k+1
              where F_i(v) is specified in Eqn. (4.3) and
                    J_Fi(v) is specified in Eqn. (4.5) ;
        }
        if ( |v_i^k+1 − v_i^k| <ε1, |F_i| <ε2 ) { /* convergence check */
            if ( v_i did not converge on last iteration ) {
                if ( v_i is active ) {
                    /* this is the selective-trace portion */
                    Schedule( v_i , t_n+1 );
                    forall ( v_j ∈ Fanout (v_i) ) Schedule( v_j , t_n );
                }
                else {
                 /* do nothing (this is the latency exploitation) */ }
            }
            else {
             /* do nothing (this breaks feedback loops) */ }
        }
        else { /* node has not converged */
            Schedule( v_i , t_n );
            forall ( v_j ∈ Fanout (v_i) ) Schedule( v_j , t_n );
        }
    } until ( |v_i^k+1 − v_i^k| <ε1, |F_i| <ε2, i =1,....,n )
}
```

■

The following definition is used above:

$$v^{k,i} = [v_1^{k+1}, v_2^{k+1}, \cdots, v_{i-1}^{k+1}, v_i^k, v_{i+1}^k, \cdots, v_n^k]^T$$

The algorithm above has two features not present in the SPLICE1.3 algorithm:

- If a node voltage does not converge, the node is rescheduled at the current time point t_n along with its fanout nodes.
- All nodes are processed until their voltages converge. When a node converges at t_n, it schedules itself at t_{n+1} and schedules its fanouts at t_n, if active. However, if it is scheduled again at t_n, by one of its fanin, and converges again, it does not schedule any additional events. This approach breaks feedback loops, since two nodes which are fanouts of each other would schedule each other indefinitely at t_n if this approach was not used.

The speed improvement obtained by the SPLICE1.7 program compared to the SPICE2 program was in the range of 5 to 50 times faster for a number of MOS digital circuits containing up to 1200 transistors [Sal83]. However, the ITA approach required approximately twice as much CPU-time to simulate a circuit compared to SPLICE1.3 which used timing simulation [Sal84]. Again, the improvements in reliability and numerical robustness far outweighed the cost of the increase in run-time.

While the converged relaxation scheme is provably better than the non-iterated approach, it is not without problems. One problem is the speed of convergence. For example, SPLICE1.7 was able to simulate accurately an NMOS operational amplifier but it required more than two times the CPU-time used by SPICE2 [NeSa83]. The circuit is a tightly-coupled analog circuit with large forward gain and capacitive feedback [Sen82] and, in this application, the node-by-node decomposition strategy used in SPLICE1.7 is inappropriate. For this same reason, convergence is also very slow in the presence of large floating capacitors and small drain and source resistors, usually found in detailed MOS transistor models. Another problem is due to nonconvergence. Since a fixed time-step is used, the program simply stopped when it was unable to converge to a

solution within a specified number of relaxation-Newton iterations. Obviously, a variable step algorithm would resolve this problem and would also allow the solutions to be computed accurately based on a local truncation error criterion. These and other problems were solved in the SPLICE2 and SPLICE3.1 programs. Since the nature of the improvements in SPLICE2 is associated with multirate integration, its description is postponed until Chapter 5.

4.5. SPLICE3.1 - GLOBAL-VARIABLE TIME-STEP ITA

A new robust version of ITA has been implemented in the SPLICE3.1 program. It differs from SPLICE1.7 in two respects:

- it uses partitioning to improve the speed of convergence for tightly-coupled circuits
- it achieves better accuracy by using a LTE-based time-step control

The SPLICE3[3] program also provides detailed MOS level 1 and MOS level 3 transistor models [Vla81], including a charge-conserving capacitance model [Yan81].

4.5.1. Circuit Partitioning

The node-based ITA approach used in SPLICE1.7 is not appropriate for circuits with tight coupling between two or more nodes, since the convergence can be very slow in this situation. One reason for this problem is that, in computing the new value for a particular node, the relaxation process effectively replaces the fanin nodes with ideal voltage sources of constant value. Therefore, the true Norton equivalent contributions from the fanin nodes are not used in the computation of a new value for the node. SPLICE2 used an improved representation of the neighboring nodes based on a current and conductance model, rather than constant voltage sources, and this approach was

[3] This new version of SPLICE is based on the library of subroutines now used in a number of simulators under development at the University of California at Berkeley, including Harmonica [Kun85], SPICE3 [Qua85], Splax [Whi85b], RELAX2 [Whi85a].

called the *coupling method* [Kle84]. This fanin node model is only approximate since the exact Norton equivalent circuit for each node is expensive to calculate for large circuits. While this approach improved the convergence speed on some examples, the technique was heuristic in nature and did not solve the general problem of coupling between more than two nodes in feedback loops.

As was realized in early mixed-level simulators such as SPLICE1, tightly-coupled subcircuits are better solved using direct methods [New78]. However, it is difficult for users to identify tightly-coupled blocks manually, especially when the degree of coupling is a function of time and hence may change over the simulation interval. A more effective approach to the coupling problem is to identify strongly-coupled components in the circuit automatically and to group them together to form subcircuits - a process referred to as *circuit partitioning*. Since the variables associated with the subcircuits are assumed to be tightly-coupled, the subcircuits can each be solved using direct matrix techniques, and the relaxation method can be applied between subcircuits. This technique has been used in conjunction with the Waveform Relaxation algorithm [Lel82, Car84, Whi85a, DeMa85, Mar85] with great success. The same approach can be used with nonlinear relaxation to improve convergence as described in Chapter 2. The static partitioning approach of the RELAX2 program [Whi85c] has been adopted in the SPLICE3 program and it is described briefly in the following.

The main goal of partitioning is to speed-up the convergence process of relaxation methods. Recall from Chapter 2 that the speed of convergence is controlled by the contraction factor, γ_∞, in the following way:

$$||x^{k+1} - x^k|| \leq \gamma_\infty ||x^k - x^{k-1}||$$

For a linear problem, this iteration factor can be computed quite easily. For example, if the linear problem $Ax = b$ is solved using the Gauss-Seidel algorithm, γ_∞ is equal to the

largest eigenvalue of the iteration matrix $[(L+D)^{-1}U]$, where $A=L+D+U$. Therefore, a two node linear circuit such as the one in Fig. 4.1, has an iteration factor (for the conductance portion only) given by:

$$\gamma_\infty = \frac{g_{12}}{(g_2+g_{12})} \frac{g_{12}}{(g_1+g_{12})}$$

A similar expression exists for the capacitance portion of the circuit. Note that if the two nodes are part of a larger circuit, the values of g_1 and g_2 are the Norton equivalent conductances seen from each node looking back into the rest of the circuit.

The partitioning algorithm makes use of the iteration factor to decide whether or not two nodes should be placed in the same subcircuit. If the factor is close to one and the nodes are solved independently, the convergence would be very slow. Therefore, the nodes should be placed in the same subcircuit. However, if the factor is close to zero, they may be placed in different subcircuits without adversely affecting the convergence speed. A threshold parameter, α, is used to decide whether or not the nodes should be solved together or separately.

A number of approximations are made in computing the iteration factors when partitioning MOS circuits. As MOS circuits are nonlinear, each nonlinear device must be replaced a linear equivalent device. Since a static partitioning strategy is used, worst-case conductance and capacitance values are used when replacing each nonlinear device with a linear one. However, the exact Norton equivalent model seen by each node cannot be computed efficiently because it involves tracing paths from each node to all other nodes in the circuit. For efficiency, the depth of the conductance and capacitance computing process is truncated whenever a MOS transistor is encountered since the conductance of a MOS transistor is zero in the worst case. With these heuristics applied, the following partitioning algorithm is obtained:

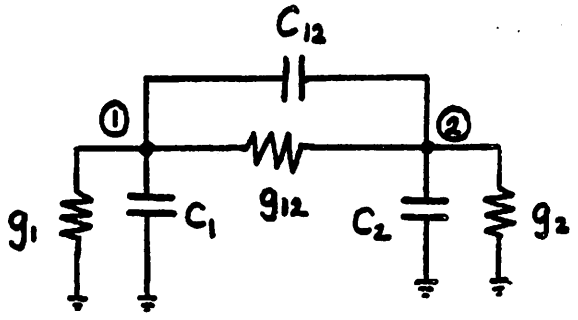


Figure 4.1 : Linear Circuit considered for Partitioning Purposes

Algorithm 4.4 (Conductance Partitioning)

foreach (*conductive element* **in** *the circuit*) {
  $g_{12} \leftarrow$ 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 g_1 and g_2, the Norton Equivalent
    conductances to ground at the two element terminals;

  if ($\frac{g_{12}}{(g_2+g_{12})} \frac{g_{12}}{(g_1+g_{12})} > \alpha$) {
    Tie the two terminal nodes together;
  }
}

A similar algorithm is used for partitioning based on capacitances. Using this approach, the run times were reduced significantly compared to the node-based approach on all examples simulated. However, the partitioning strategy described here

has a number of problems. The main problem with this approach is that it may produce unnecessarily large subcircuits since worst-case values are used in the partitioning process. The advantages of the relaxation method are lost if the subcircuits are too large. Since static partitioning is used (that is, the subcircuits are defined before the simulation begins), the latency exploitation is no longer performed at the node level but rather at the subcircuit level. All nodes in a subcircuit must be latent before the subcircuit is declared latent. While this provides a somewhat stronger condition for latency, it reduces the efficiency of the latency exploitation. Ideally, one would prefer to use small-signal conductance and capacitance values to perform the initial partitioning, and then adjust the subcircuits as these values change during the simulation. This is referred to as *dynamic partitioning* and has already been successfully applied to the simulation of Bipolar circuits using Waveform Relaxation [Mar85].

Another problem with the partitioning approach given in Algorithm 4.4 is that it is too local a criterion. For example, if two nodes are extremely tightly-coupled, relative to their coupling to neighboring nodes, they will be lumped together in the same subcircuit but the neighboring nodes may be incorrectly placed in other subcircuits. If the neighboring nodes are truly coupled to either of the two nodes, the convergence may still be slow [Whi85c]. One practical problem in partitioning is that it is a time-consuming task. Care must be taken in the definition of the data structures and partitioning algorithms so that the partitioning phase does not dominate the total run time for large circuits. This is of major concern in dynamic partitioning [Mar86].

Recently, the concept of overlapping partitions has been introduced [Mok85]. In this approach, each subcircuit is expanded to include its fanout nodes before computing a solution. Since all the subcircuits perform this operation, the expanded subcircuits overlap with each other. This technique improves the convergence speed of the relaxa-

tion method, resulting in fewer iterations than the standard partitioning strategy. The main drawback of overlapping scheme is that the amount of work associated with each subcircuit increases and, although the number of iterations is reduced, the run time may increase. One could conceive of extending this notion of overlapping partitions to include as many nodes as required, depending on the coupling in a given situation. That is, parts of two or more subcircuits could be combined if it is determined that the nodes in the subcircuits are tightly-coupled over a particular interval of time. This would allow for an arbitrary amount of overlap between subcircuits during the iterative process, and may improve the convergence speed beyond the previous scheme in [Mok85].

4.5.2. Global-Variable Time-Step Control

SPLICE3.1 uses a global-variable time-step algorithm in which the components in the system are integrated using a single common time-step. This integration time-step is selected based on the fastest changing variable in the system, the same strategy used in direct methods. However, only the active subcircuits are processed at each time point, and these subcircuits are identified using the selective-trace algorithm. The main steps in the global time-step ITA algorithm are given below following a brief description of the notation to be used.

Notation for Algorithm 4.5: (see Fig. 4.2)

Assume that a given circuit is partitioned into n subcircuits $S_1, S_2, \cdots, S_i, \cdots, S_n$. The i th subcircuit, S_i, has n_i internal variables and n_e external inputs. The internal variables given by int(S_i) = { $x_1, x_2, \ldots, x_{n_i}$ } are those variables computed whenever subcircuit S_i is processed. They are defined in vector form as $v_i = [x_1, x_2, \cdots, x_{n_i}]^T$. The external inputs of a subcircuit are other nodes which affect the internal nodes of the subcircuit. They are specified as $Fanin(S_i) = \{ e_1, e_2, \cdots, e_{n_e} \}$. The fanouts of a

subcircuit are associated with the internal nodes of the subcircuits. Hence, the set of subcircuits affected by an internal node, x_j, and are specified as $Fanout(x_j) = \{ S_1, S_2, \cdots, S_k \}$. The following definition is also used:

$$v^{k,i} = [v_1^{k+1}, v_2^{k+1}, \cdots, v_{i-1}^{k+1}, v_i^k, v_{i+1}^k, \cdots, v_n^k]^T .$$

Algorithm 4.5: Global-Variable-Time-Step ITA

```
Partition();
t_n ← 0;  h_min ← h_start ;
while ( t ≤ T_stop ) {
    stepRejection = FALSE;
    h_next ← h_min; t_n ← t_n + h_next ; h_min ← h_max;
    foreach ( input i at t_n ) {
        if ( e_k is active ) {
            forall ( S_j ∈ Fanout (e_k )) Schedule( S_j , t_n );
        }
    }
    repeat {
        foreach ( event i at t_n ) {
            solve J_F_i (v^{k,i})(v_i^{k+1} − v_i^k) = − F_i (v^{k,i}) for v_i^{k+1}
              where F_i (v ) is specified in Eqn. (4.3) and
                    J_F_i (v ) is specified in Eqn. (4.5)
        }
        if ( ||v_i^{k+1} − v_i^k|| < ε_1, ||F_i|| < ε_2 ) {  /* convergence check */
            if ( v_i did not converge on last iteration ) {
                foreach ( x_i ∈ int (S_i ) ){
                    if ( x_i is active ) {
                        if ( CheckAccuracy( x_i ) = TRUE ) {
                            h_i ← pickStep( x_i ); h_min ← min( h_min, h_i );
                            schedule( x_i , t_n+1 );
                            forall ( S_j ∈ Fanout (x_i ) ) schedule( S_j , t_n );
                        }
                        else {  /* solution not accurate enough */
                            t_n ← t_n − h_min; h_min ← h_min/2; stepRejected = TRUE;
                        }
                    }
                }
            }
        }
        else {  /* subcircuit has not converged */
            if ( itercnt > maxitercnt ) {  /* solution not accurate enough */
                t_n ← t_n − h_min; h_min ← h_min/2; stepRejected = TRUE;
            }
            else {
                Schedule( S_i , t_n );
                foreach ( x_i ∈ int (S_i ) ) {
```

```
                    if ( x_i is active )
                        forall ( S_j ∈ Fanout (x_i ) ) Schedule( S_j , t_n ):
                }
            }
        }
    } until (( ||v_i^{k+1} − v_i^k|| <ε_1, ||F_i|| <ε_2, i =1,....n ) OR (stepRejection))
}
```

■

In the algorithm above, the *CheckAccuracy*(x) routine uses a local truncation error criterion to determine if the computed solution for x is accurate and, if so, returns "TRUE". The *PickStep*(x) routine uses a LTE estimate to pick the next recommended step size for x.

The main differences between this algorithm and the one used in SPLICE1.7 are due to the actions taken when the subcircuit variables converge at a time point and when they do not converge in a specified number of relaxation-Newton iterations. When the active subcircuits converge at a time point, t_n, the local truncation errors for their internal variables are estimated [Bra72] and the new global time-step, h_{next}, is set to the smallest recommended step in the system, $h_{\min}$. If the accuracy in the solution computed at t_n is unacceptable, the solution is rejected and the integration is retried with the smaller time-step. Similarly, if the iterations do not converge within a specified number of iterations, the time-step is rejected and a smaller step is used.

4.6. LATENCY AND EVENT SCHEDULING

4.6.1. Latency Detection

The most critical aspect in ITA, in terms of accuracy, is the detection of the latency condition. For example, if component x is identified as being latent prematurely, any small errors in its value will be propagated to the other components producing errors in their solutions. If the component is thought to be latent but, in reality, it is changing very slowly the results may be completely wrong. So the overriding

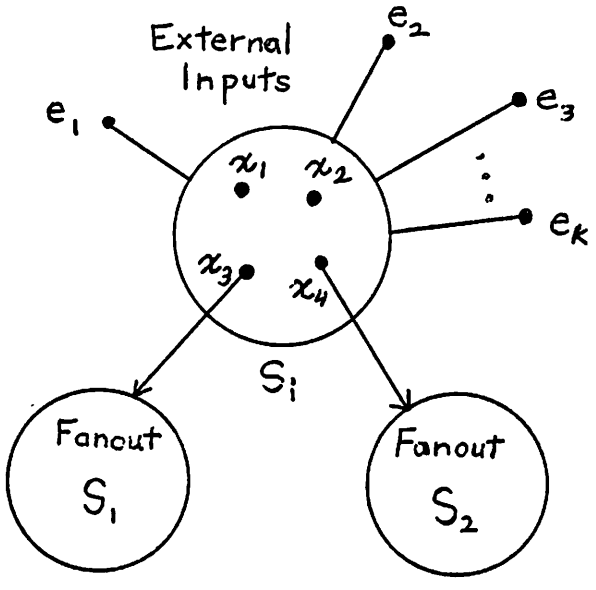


Figure 4.2 : Notation associated with Subcircuits

question is: how can one be sure that a variable has reached a steady-state value? The simplest approach is to test if the following condition is satisfied:

Latency Condition 1:

$$| x_{n+1} - x_n | < \epsilon_v \tag{4.6}$$

where $x_{n+1} = x(t_{n+1})$, $x_n = x(t_n)$ and ϵ_v is some small number. As illustrated in Fig. 4.3, the component is considered latent if the difference in the computed solution at two successive time points is less than some pre-specified amount, ϵ_v. For a fixed time-step ITA algorithm [Sal83], this is reasonable check as long as ϵ_v is specified properly and one additional check is done, as described shortly. There are situations where Condition 1 may fail, as shown in Fig. 4.4, where the true solution rises and then falls before

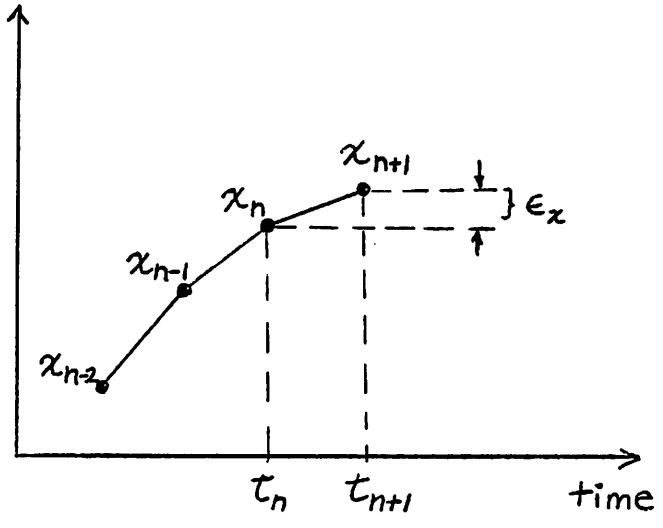


Figure 4.3 : Simple Latency Detection

reaching a steady-state value. If the time points are chosen such that Condition 1 is satisfied, latency will be detected incorrectly. A more conservative version of Condition 1 requires that the inequality be satisfied for two time points that are not adjacent.

Latency Condition 1.1:

$$| x_{n+k} - x_n | < \epsilon_x \quad , \quad k \geq 1 \tag{4.7}$$

While this conservative approach works well in practice, it is still not strong enough to handle the general case. For example, if a global variable time-step control is used, the step sizes may be very small due to some fast component resulting in small changes in x over a large number of time points (if x is a slower component). In this case, it would make more sense to use a rate-of-change criterion to detect latency rather than

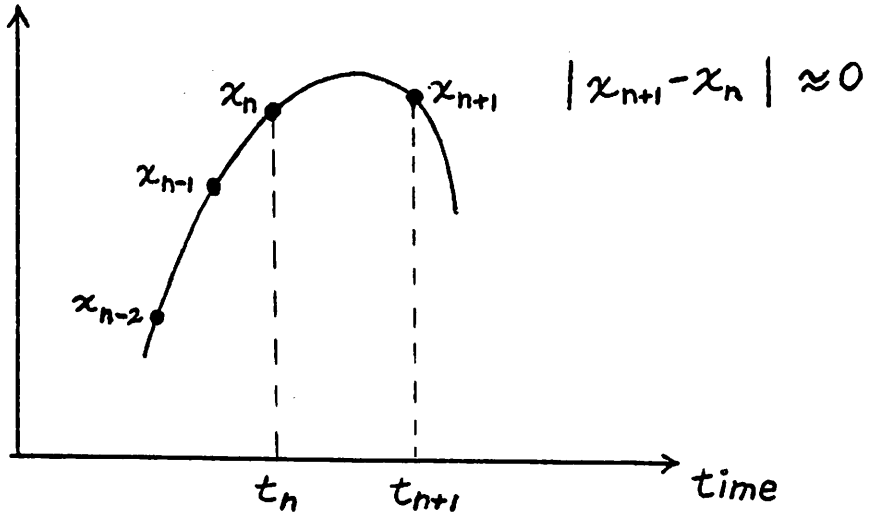


Figure 4.4 : Potential Problem in Latency Detection

the absolute change in x . That is, use the check

Latency Condition 2:

$$\frac{|x_{n+1} - x_n|}{h_n} < \epsilon_{\dot{x}} \tag{4.8}$$

As shown in Fig. 4.5, this requires that $\dot{x} \approx 0$ to satisfy the latency condition. This method also encounters problems with the example in Fig. 4.4 since $\dot{x} \approx 0$ as the signal switches direction. A more conservative way to do this type of latency check would be to use the strategy of Condition 1.1 and include a number of points from the past.

Latency Condition 2.1:

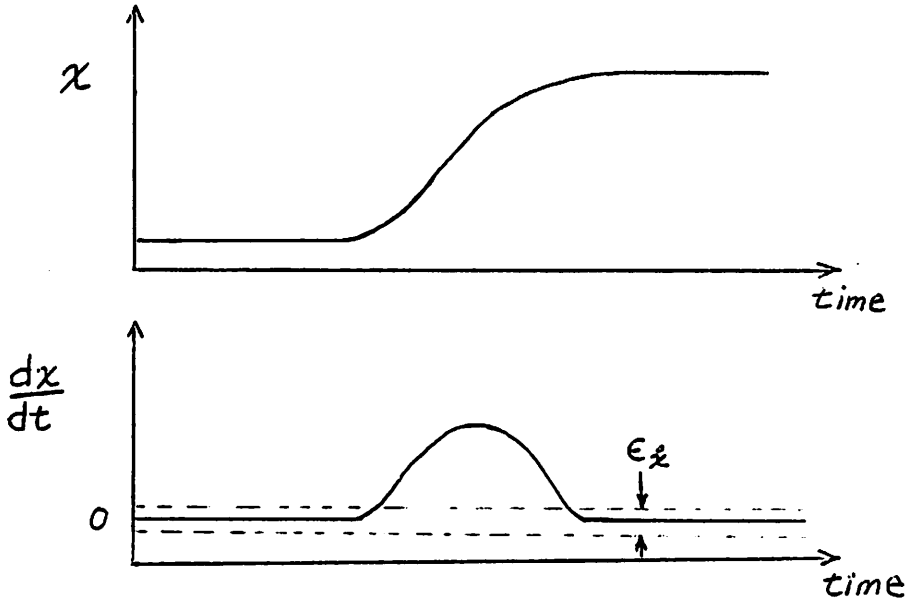


Figure 4.5 : Variable Step Latency Criterion Based on Rate-of-change

$$\frac{1}{k}\sum_{j=1}^{k}\frac{|x_{n+2-j}-x_{n+1-j}|}{h_{n+1-j}} < \epsilon_i \quad , \quad k \geq 1 \tag{4.9}$$

This condition uses an average rate of change based on the previous k solutions to detect latency and this overcomes the problem given in Fig. 4.4. However another problem arises if the true value of $\dot{x}$ is some small non-zero value that eventually changes the value of x significantly at some point in the future. To resolve this problem, a "wake-up" mechanism should be used with either Condition 1.1 or 2.1 when it is anticipated that component x has undergone a significant change in value. That is, the actual rate-of-change of x should be used to predict the wake-up time point, as follows:

Wake-up Condition 1:

$$h_{next}\,\frac{|x_{n+1}-x_n|}{h_n} > \epsilon_x \tag{4.10}$$

and $t_{wake-up} = t_{n+1} + h_{next}$. This wake-up condition can be used to compute h_{next} and the component should be re-activated and solved at $t_{wake-up}$. This process is illustrated in Fig. 4.6.

The latency and wake-up conditions specified above work well in practice and their use can be justified by considering latency exploitation as the use of a zeroth-order explicit integration method as described in reference [Rab79]. Explicit integration algorithms are obtained directly from a Taylor series expansion of the solution at the point t_n:

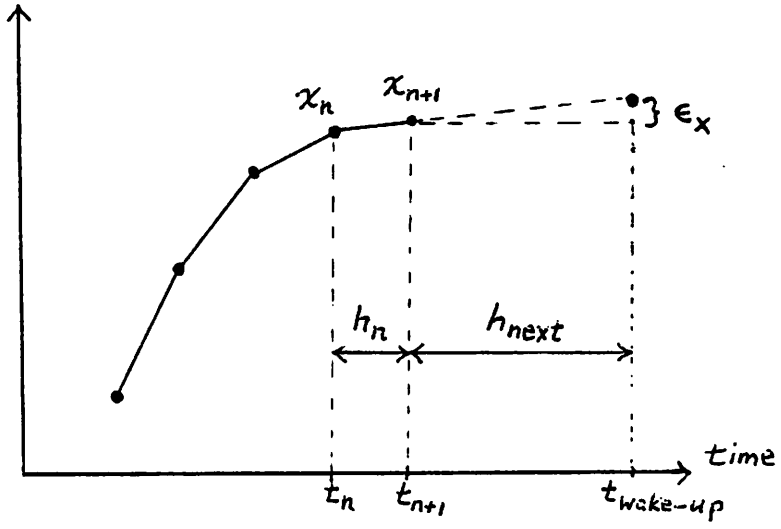


Figure 4.6 : Wake-up Mechanism

$$x_{n+1} = x_n + h_{n+1}\dot{x}_n + \frac{h_{n+1}^2}{2}\frac{d^2x_n}{dt^2} + \cdots \tag{4.11}$$

A zeroth-order method uses only the first term and produces the following trivial integration method for which $x(t_{n+1})$ is simply updated with the value $x(t_n)$ at the previous time point:

$$x_{n+1} = x_n \tag{4.12}$$

This integration method has a local truncation error (LTE) given by:

$$LTE = h_{n+1}\dot{x}(\xi) \qquad t_n \leqslant \xi \leqslant t_{n+1}$$

An estimate of the LTE can be obtained using a finite difference approximation for $\dot{x}$:

$$\dot{x}_n(\xi) \approx \frac{x_{n+1} - x_n}{h_n}$$

Therefore the LTE estimate is given by:

$$LTE \approx h_{n+1}\frac{x_{n+1} - x_n}{h_n}$$

A check for latency can now be constructed from this analysis. The integration method specified in Eqn. (4.12) can be used whenever the following condition is satisfied:

Latency Condition 3:

$$h_{n+1}\frac{|x_{n+1} - x_n|}{h_n} < E_{user\,LTE} \tag{4.13}$$

where $E_{user\,LTE}$ is the allowable local truncation error specified by the user.

For a fixed time-step algorithm, this latency check is equivalent to Condition 1 since $h_n = h_{n+1}$ for all n. Of course, the value for ϵ_v in Condition 1 must be derived the same way as $E_{user\,LTE}$ to be identical to Condition 3. For a variable step algorithm, one could rewrite Condition 3 as:

$$\frac{|x_{n+1} - x_n|}{h_n} < \frac{E_{user\,LTE}}{h_{n+1}}$$

By replacing h_{n+1} with a constant value of step size h_{max} such that $h_{max} >> h_{n+1}$, one

can provide a somewhat tighter constraint:

$$\frac{|x_{n+1} - x_n|}{h_{n+1}} < \frac{E_{userLTE}}{h_{\max}}$$

Then Latency Condition 2 and 3 can be made identical by setting $\epsilon_{\dot{x}} = \frac{E_{userLTE}}{h_{\max}}$. Note that Condition 3 is an *a posteriori* criterion (i.e., it is used after selecting h_{n+1}) to detect latency. A similar criterion can be used in an *a priori* manner to decide when to activate the component. The idea is to use the LTE requirement to predict the time point when the zeroth-order integration method is no longer valid by checking when Latency Condition 3 is violated:

$$h_{new} \frac{|x_{n+1} - x_n|}{h_n} > E_{userLTE} \tag{4.14}$$

where $h_{new} = t_{wake-up} - t_{n+1}$ and $t_{wake-up}$ is the time when the component should be activated. This wake-up time can be computed as follows:

$$t_{wake-up} = t_{n+1} + \frac{E_{userLTE} h_{n+1}}{x_{n+1} - x_n} \tag{4.15}$$

and this is identical to Wake-up Condition 1. Therefore, the intuitive arguments which lead to Latency Conditions 1 and 2, and Wake-up Condition 1 are well-supported by the above analysis.

4.6.2. Electrical Events and Event Scheduling

The next issue is to define precisely the notion of electrical events for use in conjunction with the scheduling algorithm. The proper definition of this concept is important from the standpoint of efficiency and accuracy, as will be seen. In logic analysis, an event occurs when a node makes a transition from one state to another (different) state. The event causes the fanouts of the node to be scheduled in the time queue. As long as the node remains in the same state, no additional events are generated. Since

logic states are discrete, logic events are easy to identify. In electrical analysis, there is a continuum of "allowed states" making it more difficult to distinguish a significant event from an insignificant one. However, the definition of logic events can be extended in a straight-forward manner to electrical analysis. The resulting definition of an electrical event is connected with the notion of "active" and "latent" components.

Definition 4.2 : (Electrical Events)

In electrical analysis, a component is "latent" if it satisfies one of the latency conditions given by Eqns. (4.6-4.9). Otherwise, it is an "active" component making a transition from one electrical value (or state) to another. Active components generate electrical events each time they make a transition to a new value. ■

The usefulness of this definition is seen in the following. Consider the two-stage inverter of Fig. 4.7. For this circuit, $A \in Fanout(I)$ and $B \in Fanout(A)$. As depicted by the arcs in the corresponding graph, there are four ways to schedule nodes:

(1) node I can schedule node A (fanout scheduling)
(2) node A can schedule node A (self-scheduling)
(3) node A can schedule node B (fanout scheduling)
(4) node B can schedule node B (self-scheduling)

Whether a given node (say, node A) should actually schedule any events depends on its own state and the state of its fanouts (node B in this case). Since each node can be either "active" or "latent", a total of four cases exist. These cases are listed in Table 4.1 along with the recommended action to be taken by node A for each case.

As the table indicates, case (2) is the only case where the scheduling mechanism is conservative. The other cases do not introduce any additional work or create accuracy problems and therefore are listed as *reasonable*. However, case (2) can be a source of either accuracy problems or excessive computation. To see this, consider the circuit in

case	status of node A	status of node B	action by node A	comment
(1)	active	active	schedule self at t+h schedule fanouts at t	reasonable
(2)	active	latent	schedule self at t+h schedule fanouts at t	conservative
(3)	latent	active	no scheduling req'd	reasonable
(4)	latent	latent	no scheduling req'd	reasonable

Table 4.1 : Four cases in electrical event scheduling

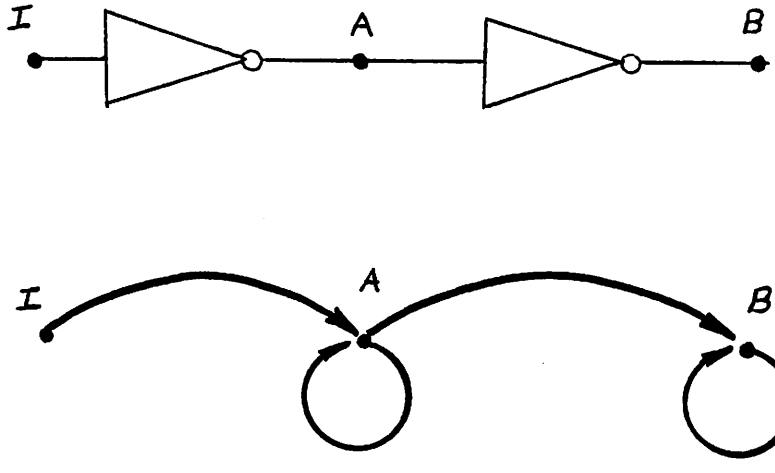


Figure 4.7 : Scheduling possibilities for a simple example

Fig. 4.8. If node A is "active", it will force nodes B, C and D to be processed if the action recommended in Table 4.1 is taken. In reality, only node B should be processed. The other two nodes do not change due to the bias conditions, but this is not known *a priori*. Therefore, case (2) is considered to be a conservative scheduling strategy. The

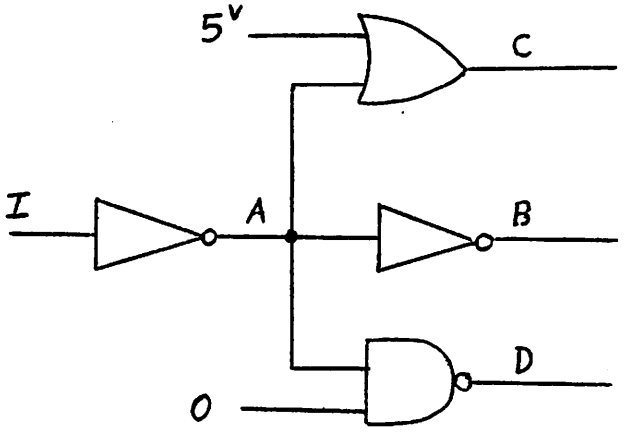


Figure 4.8 : Conservative scheduling case

alternative would be to ask the question: is fanout x_j *sensitive* to changes in x_i? Here, x_i = A and *Fanout* (x_i) = { B , C , D }. Only an affirmative response to this question causes a particular x_j to be scheduled by x_i. Otherwise x_j should not be scheduled.

The conditions associated with case (2) can also be viewed as a wake-up condition due to inputs. That is, "Does the change at node A wake-up node B?". The previous wake-up conditions were all handled via the self-scheduling mechanism. In this case, the question is whether or not a change at x_i translates to a change at a fanout x_j such that x_j violates its latency condition. Since x_j may have a number of fanin nodes which are active, superposition must be used to determine the combined effect of all active fanin nodes on x_j. This involves determining the transconductance, $\frac{\partial f_j}{\partial x_i}$, and performing the computation:

$$\Delta x = \frac{h_n}{C_j} \sum_{i=1}^{k} \frac{\partial f_j}{\partial x_i} \Delta x_i \tag{4.16}$$

where k is the number of fanin nodes of x_j which are active. h_n is the current step size, and C_j is the total capacitance at node x_j. This computation assumes that all the additional current, due to changes in the fanin nodes, act to charge the capacitances at node x_j. This produces a new wake-up condition due to the inputs, as follows:

Wake-up Condition 2:

$$h_{new} \frac{| x_{n+1} - x_n |}{h_n} + \Delta x > \epsilon_x$$

where $h_{new} = t_{new} - t_{latent}$, and t_{new} is the current time point. In the worst-case, the computation in Eqn. (4.16) can be as expensive as performing an evaluation of x_j, but it certainly is not as accurate. Since there is no way to guarantee that Wake-up Condition 2 is a sufficient check for latency violation, since is only a local criterion, it is better to perform the evaluation of x_j rather than the sensitivity check to guarantee that an error is not made inadvertently. This results in a stronger condition for latency, which involves the fanin nodes also being latent.

The ideas presented above are formalized in the following:

(1) A component x_i is defined as being latent if

(a) it satisfies the latency conditions specified in Eqns. (4.6-4.9) and

(b) all $e_k \in Fanin(x_i)$ satisfy their latency criteria

(2) A latent component does not generate any events.

(3) If a component is not latent, then it is active and hence will generate events for itself and for all $x_j \in Fanout(x_i)$ after every transition.

(4) A latent component x_i is scheduled for re-evaluation if

(a) the wake-up condition specified in (4.10) is satisfied, or

(b) any component $e_k \in Fanin(x_i)$ becomes active

4.6.3. Latency in the Iteration Domain

Another form of latency can be exploited at each time point due to the decoupled nature of the relaxation process. Since the components in the system are changing at different rates, it is quite possible that slowly varying components will converge quickly at each time point since their behavior can be predicted accurately. Once these components have converged, there is no need to reprocess them at the same time point unless required to do so by some other component. This form of latency is called iteration domain latency and can also be exploited efficiently using the same event-driven techniques used for time domain latency.

The iteration domain is a discrete space in which a sequence of iteration values of a component can be represented as a function of the iteration number [Kle84]. This iteration domain can be viewed in the same way as the time domain. For example, if a converging sequence of iterations for a component, x_i, is plotted against the iteration number, a waveform is produced as shown in Fig. 4.9. The detection of latency in the time domain is seen to be analogous to the detection of convergence in the iteration domain. In fact, since the "step size" is fixed in the iteration domain, the check for convergence should be similar to the Latency Condition 1 given earlier. This corresponds to to checking if the iteration waveform is "flat enough" [Kle84] and is given as:

Convergence Criteria 1:

$$|x_i^{k+1} - x_i^k| < \epsilon$$

which is consistent with the usual check for convergence. False convergence occurs when the condition is satisfied but the necessary accuracy has not been obtained. Therefore, a check similar to Latency Condition 1.1 would be better to avoid this problem

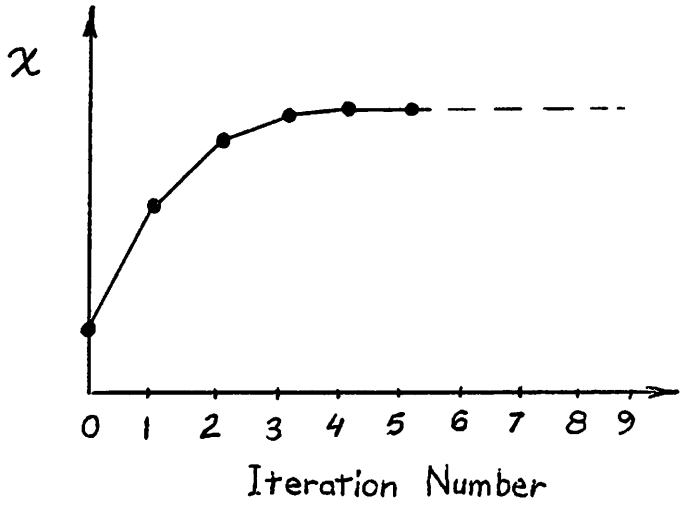


Figure 4.9 : Iteration domain waveforms

[Kle84].

Convergence Criteria 1.1:

$$|x_i^{k+m} - x_i^k| < \epsilon, \quad m > 1$$

To exploit latency in the iteration domain using event-driven techniques, a table similar to the one for latency in time is necessary. In the iteration domain, if a component is "iterating" it is equivalent to being "active" in the time domain, and if it has "converged" in the iteration domain, it is equivalent to the "latent" condition in the time domain. Note that latency in time implies latency in the iteration domain, but latency in the iteration domain (i.e., convergence) *does not* imply latency in time. In fact, when a component converges in the iteration domain, a separate test is necessary to determine if it is active or latent in the time domain. The four cases in the iteration domain are

listed in the table below along with the recommended action for node A, assuming that node A is in the "converged" state initially and enters the state listed in column 2 after computing its new value.

case	new status of node A	status of node B	action by node A	comment
(1)	iterating	iterating	schedule self at t schedule fanouts at t	reasonable
(2)	iterating	converged	schedule self at t schedule fanouts at t	conservative
(3)	converged	iterating	no scheduling req'd	reasonable
(4)	converged	converged	no scheduling req'd	reasonable

Table 4.2 : Four cases in iteration domain latency

The table shows that case (2) is again the only conservative scheduling situation. To understand this case, consider Fig. 4.8 again. Each time node A performs an iteration, it will schedule nodes B, C and D. However, as before, only node B should be processed as nodes C and D are latent in time and hence are in the converged state at the time point. If node A requires many iterations to converge, it will schedule nodes C and D many times resulting in a lot of unnecessary work. However, there is no need to repeatly schedule all its fanouts on every iteration, especially since the nodes have a self-scheduling ability. Therefore, one strategy might be for node A to schedule its fanouts *on every other iteration* rather than on every iteration. This could be used for both case (1) and case (2) since the self-scheduling mechanism would take care of any additional scheduling of node B.

4.7. SIMULATION RESULTS

4.7.1. Speed Improvement Due to Latency Exploitation

Table 4.3 contains simulation results from ITA in SPLICE3.1 for a number of circuits in Table 3.1. Three simulations were performed on each circuit and are listed with the suffixes ".1", ".2" and ".3" respectively. In the first simulation, latency was not exploited in any form. In the second simulation, only iteration domain latency was exploited. In the third case, both time domain and iteration domain latency were exploited. The circuit name and number of nodes are given in the first two column. The CPU-time, number of Newton iterations and number of time points are provided in the next three columns. The number of rejections due to nonconvergence and LTE are listed in the 6th and 7th columns. Note that very few rejections are due to nonconvergence of the relaxation iterations. In practice, the constraints imposed by the LTE time-step requirements are usually much stronger than those imposed by the diagonal dominance requirement. The last column shows the improvement factor in the run time when iteration domain latency and time point latency are exploited. These results indicate the importance of exploiting both forms of latency. As one would expect, the improvement increases as the size of the circuit increases.

4.7.2. Global-Variable Time-Step ITA vs. Direct Methods

In Table 4.4, SPLICE3.1 using global-variable time-step control is compared to direct methods used in SPLICE3.1 and SPICE2G.6. The results indicate that the speed-up over the SPICE2G.6 program is substantial in all cases. However, the speed-up over the direct methods available in SPLICE3.1 is not very large compared to the ideal speed-up computed earlier in Chapter 3. For the OPAMP circuit, SPLICE3.1 is as fast as direct methods. However, in this case the partitioning algorithm places all nodes into

circuit	size	CPU-time (sec.)	iters	timepts	numnon converg	numlte reject	speed-up (time)
decpla.1	56	43.88	12784	141	3	27	1.0
decpla.2		30.56	8621	141	3	27	1.4
decpla.3		26.64	7447	132	4	21	1.9
cramb.1	149	414.33	163392	612	36	123	1.0
cramb.2		194.83	65590	615	37	124	2.1
cramb.3		122.08	37520	611	33	129	3.4
scdac.1	154	289.27	64989	505	15	74	1.0
scdac.2		204.35	41599	505	15	74	1.4
scdac.3		176.37	33566	501	15	74	1.6
ckt3.1	312	931.87	351260	758	62	143	1.0
ckt3.2		425.33	144305	757	58	143	2.2
ckt3.3		304.09	104751	764	58	157	3.0
eprom.1	630	9530.23	1042500	1756	108	353	1.0
eprom.2		4038.46	339431	1760	111	352	2.4
eprom.3		2374.43	155007	1713	111	326	4.0

Table 4.3 : Simulation Results from SPLICE3.1 - Legend for suffix on circuit names: 1=no latency exploitation; 2=only iteration domain latency exploited; 3=time point latency and iteration domain latency; CPU-time is given in sec. on VAX-8650; iters=number of relaxation-Newton iterations; numnonconverg=no. of iteration count rejection; numltereject=number of LTE rejections; speed-up =improvement in speed compared to no exploitation case.

Circuit	Size (nodes)	Ideal Speed-up	Actual Speed-up (SPLICE3.1)	Actual Speed-up (SPICE2G.6)
OPAMP	13	1.8	1.0	13.0
uP Control	56	2.2	1.0	9.5
CRAMB	149	12.3	2.6	11.3
SCDAC	154	4.1	1.2	12.4
CKT3	312	6.5	1.5	5.0
DIGFI	378	8.2	2.6	10.7
EPROM	630	18.4	2.7	12.0

Table 4.4: ITA vs. Direct Methods

one subcircuit which forces SPLICE3.1 to use direct methods. Therefore this circuit is a

special case where direct methods should be used. For the larger circuits, the performance of global time-step ITA was better due to the amount of waveform latency but certainly not as efficient as expected.

There are a number of reasons why the program does not reach the ideal speed improvements over direct methods. The first reason is that the time-step selection is conservative. If the step is too large, the relaxation method may not converge or the LTE criterion may not be satisfied. To reduce the chances of a step rejection, the time-step used is smaller than the one allowed by the LTE estimation. A second reason for the inefficiency is that the program is not exploiting latency at the node level but rather at the subcircuit level. Node-by-node decomposition would allow efficient latency exploitation but would adversely affect the convergence speed of relaxation method and therefore cannot be used. A third reason stems from the conservative scheduling strategy due to case (2) in Tables 4.1 and 4.2. This is probably a major reason for the limited speed improvement, but must be used to guarantee that an error is not made in latency detection. Another important reason for the inefficiency is that coupling between subcircuits changes during the simulation but the subcircuits do not follow change since static partitioning is used. This effectively increases the number of relaxation iterations at each time point. As a result, the cost of each solution point may be much higher than direct methods even though fewer points are computed for each waveform due to latency exploitation. Therefore, the partitioning algorithm determines to a large extent the speed-up obtained using nonlinear relaxation.

4.8. CONCLUSIONS

A number of nonlinear relaxation algorithms have been described in this chapter. It was shown that nonlinear relaxation combined with event-driven selective-trace techniques can be used to exploit the latency property of large circuits. This technique,

called Iterated Timing Analysis (ITA), has been implemented in the SPLICE family of programs including a new version in SPLICE3.1. An event scheduling algorithm for electrical analysis was also described in this chapter. It was shown to depend critically on the proper detection of the latency condition. A number of latency detection criteria were presented along with wake-up mechanisms to reduce the effect of errors in the latency detection. Using the ITA approach, large speed improvements were obtained over the SPICE2G.6 program, specifically, 5 to 15 times faster. However, the improvements were not as encouraging when compared to the direct methods used in SPLICE3.1. The inefficiency was attributed to the conservative scheduling algorithm and the static partitioning strategy.

The time-step control used in the SPLICE programs described in this chapter can be viewed as a limited multirate approach since each component may use a time-step which either common to all active components or a time-step which is independent of other components but based on a latency criterion. The actual step sizes during the latent periods are determined by the wake-up conditions. However, when a component is active, its time-step is determined by the fastest changing variable in the system and not necessarily by its own rate of change. A true multirate scheme in which different components are permitted to use different step sizes would improve the efficiency of the simulation. It would also resolve the problem of latency detection since the components that are latent would still perform an integration operation but the step size would be much larger than the other components. A number of such schemes are presented in the chapters to follow.

CHAPTER 5

EVENT-DRIVEN MULTIRATE INTEGRATION ALGORITHMS FOR ITA

5.1. INTRODUCTION

A multirate integration method is one that uses different step sizes to solve different variables in a system of ordinary differential equations. In the previous chapters, strong arguments were presented in favor of using integration methods which exploit the multirate nature of MOS integrated circuits to reduce circuit simulation run times. Relaxation methods provide an opportunity to use multirate methods since they decouple the variables in the system. Ideally, one would prefer to solve each component in the system separately to achieve "full" multirate integration. However, the non-zero coupling between some components requires the use of partitioning to group tightly-coupled components together into subsystems. Direct methods are used to solve each of the subsystems and the relaxation method is applied between subsystems. While the use of static partitioning limits the efficiency of multirate integration somewhat, each subsystem can still use the largest time-step that accurately reflects the behavior of its associated state variables.

The SPLICE1.7 program (which uses a fixed time-step) and the SPLICE3.1 program (which uses a global-variable time-step) both use nonlinear relaxation and event-driven techniques to exploit waveform latency, but do not exploit multirate behavior. That is, the selective-trace ITA algorithm takes advantage of a system for which most of the variables remain at an equilibrium state but does *not* take full advantage of a system for which the state variables have different rates of motion, but are not at equilibrium.

The WR algorithm *intrinsically* allows different time-steps to be used for different subsystems since the system of equations are decoupled at the differential equation level. However, each iteration in WR involves solving nonlinear differential equations. These are usually solved using an implicit integration method and, at each time point, a nonlinear algebraic system is solved using the Newton-Raphson method. Hence, each waveform relaxation iteration is rather expensive. Two factors, which are not necessarily unrelated, affect the number of WR iterations used in a simulation: the effectiveness of the system partitioning and the window selection strategy [Whi85c]. Convergence may be slow if there is moderate or tight coupling between different subsystems or if the windows are too large. If a moderately coupled system does not exhibit multirate behavior, then the global-variable time-step ITA algorithm will be more efficient, because in ITA the computational cost of performing an iteration is lower than in WR. For ITA, only one Newton iteration is performed for each relaxation iteration. While the ITA method is also prone to excessive iterations if the subsystems are moderately coupled, there is no additional penalty due to windows since they are not used.

Two approaches can be pursued to improve the multirate integration efficiency of relaxation methods. The first is to extend the ITA method to perform multirate integration while retaining its inherent advantages and the second is to reduce the computational cost of each iteration in the WR method. The first approach is explored in this chapter while the second approach is addressed in the next chapter. In particular, a new multirate scheme for ITA is described in this chapter with a method for limiting the effect of a step rejection. A new incremental repartitioning strategy is also proposed to obtain full multirate integration.

The basic concepts of event-driven multirate integration are described in Section 5.2. Previous event-driven schemes are reviewed in Section 5.3. In Section 5.4, the multirate integration scheme for ITA is described in detail along with the step rejection technique. In Section 5.5, an incremental partitioning approach is proposed for use with ITA. Simulation results from SPLICE3.2, which uses the multirate ITA scheme, are presented in Section 5.6 and compared to standard WR in RELAX2.3. Conclusions are given in Section 5.7.

5.2. BASIC CONCEPT IN EVENT-DRIVEN MULTIRATE METHODS

In this section, the basic concepts of event-driven or incremental multirate methods are described. The term *event-driven multirate integration* is used here in contrast to the way that Waveform Relaxation achieves multirate integration. In the event-driven case, an interval of time is specified for the integration process. A number of time points, or *grid points*, are defined within this interval. Components in the system are "scheduled" as events at the grid points, based on their recommended step sizes, and processed when the simulator reaches the time at which each event is scheduled. The basic idea in the event-driven approach is to keep the different components at approximately the same point in time during the integration process. In WR, the waveforms for the variables in a single subsystem are computed in the *entire* interval of interest before considering another subsystem.

This basic difference between the two approaches is illustrated further in Fig. 5.1. If WR is used to solve the two-stage inverter chain, the solution would be computed in the order shown in Fig. 5.1(a) whereas the event-driven multirate method would generate the solution in the order given in Fig. 5.1(b). In fact, for this example, the two approaches can be viewed simply as a reordering of the computation. However, there are two other key differences between the the event-driven approach and WR which are

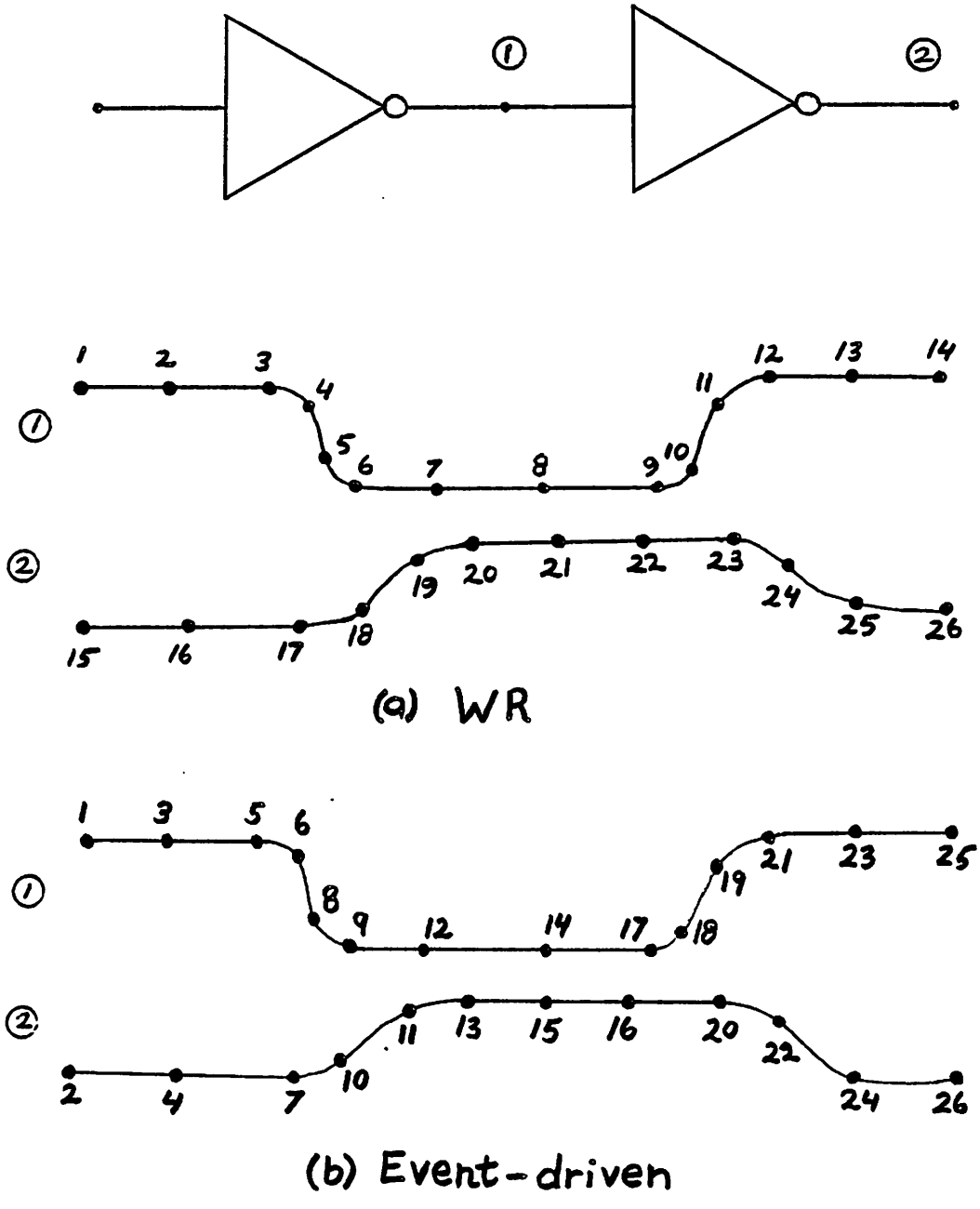


Figure 5.1 Order of Computation for WR and Event-driven Approaches

illustrated using Fig. 5.2. In this figure, three variables are being integrated between t_0 and t_4. The arcs depict the step sizes to be used during the integration process for each variable. Assume that the variables are related by the following differential equations:

$$\begin{aligned} \dot{x}_1 &= f_1(x_1, x_2, t) \quad x_1(t_0)=x_{10} \\ \dot{x}_2 &= f_2(x_1, x_2, x_3, t) \quad x_2(t_0)=x_{20} \\ \dot{x}_3 &= f_3(x_2, x_3, t) \quad x_3(t_0)=x_{30} \end{aligned} \tag{5.1}$$

Further assume that the events are processed in a time-ordered fashion. That is, all events at t_1 are processed first, then all events at t_2 are processed, and so on until t_4 is reached.

In computing $x_1(t_1)$, the question arises as to what value should be used for variable $x_2(t_1)$. This type of question comes up frequently since, in general, different

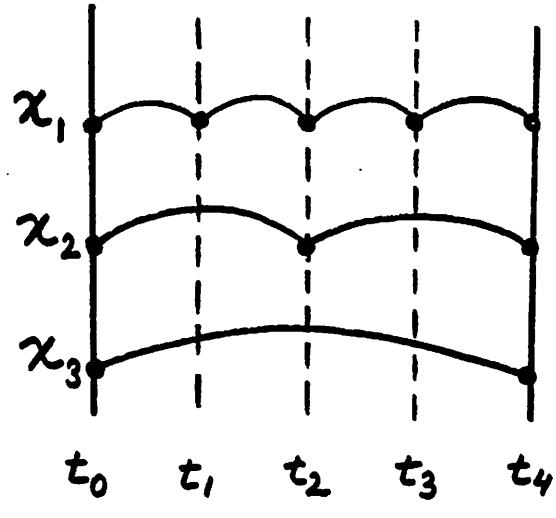


Figure 5.2 : Recommended Step sizes for a Multirate Integration Method

components may be at a different points in time during the integration process due to a time-domain decoupling of the variables. Since the behavior of x_2 in the interval $[t_0, t_2]$ is not known, a *dormant model* [Sak81] must be defined for the variable.

Definition 5.1: A dormant model, given by $x_i^D(t)$, $t \in [t_{begin}, t_{end}]$, is a representation of the behavior of a component x_i in the interval $[t_{begin}, t_{end}]$ which may be used by other components when computing their solution. A dormant model is termed "exact" if represents an accurate or closed form solution to the differential equation describing the component in the specified interval. Otherwise it is termed "approximate".

■

One of the differences between the event-driven approach and WR is the effect of time-domain decoupling on the accuracy of the solution. In WR, the dormant model for x_2 is simply its waveform from the previous iteration. Whenever a value at a particular time point is required, it is interpolated from the waveform supplied by this dormant model. In the event-driven case, it is not obvious how to construct a reasonable dormant model for x_2. One option is to use extrapolation to predict the value of $x_2(t_1)$ based on its values at previous time points. This extrapolated value can be used to compute the value of $x_1(t_1)$. Unfortunately there is no guarantee that $x_2(t_1)$, obtained in this manner, is correct since its true solution has not be computed. As a result, an error is introduced into the value computed for $x_1(t_1)$, called a *time-domain decoupling error*. This decoupling error is propagated to other components that depend on x_1 and may eventually affect the accuracy of the overall solution. Therefore, in the event-driven approach, the nature of this decoupling error must be well understood and error propagation of this type must be controlled. In WR, the decoupling error decreases with each iteration and is reduced to a simple interpolation error when the waveforms converge to their final solution.

A second difference between WR and the event-driven approach is the effect of step rejections. In WR, a step rejection during the integration process is not a major concern since it does not affect other components. If a step rejection occurs for a particular variable, the solution is simply retried with a smaller step and the integration process continues normally to the end of the time interval. The solution waveform is used as a dormant model for the solution of neighboring components with all rejection information filtered out, as it is unimportant to other components. In the event-driven case, a step rejection could have a significant impact on the performance. For example, if x_3 rejects the solution obtained at t_4, the component must be re-integrated using smaller steps over the same interval. However, in generating the solutions for $x_1(t)$ and $x_2(t)$, $t \in [t_0, t_4]$, there was an overriding assumption that the values for x_3, provided by its dormant model, were correct in that interval. When the solution $x_3(t_4)$ is rejected, it implies that the dormant model was not valid. As a result, there may be large errors in the solution for x_2 and indirectly in the solution for x_1 through f_1. It now becomes necessary to "roll back" the solutions for x_1 and x_2 and to retry the whole process. Therefore, the CPU-time used to generate the previous solutions is essentially wasted. Clearly, an efficient roll-back scheme is very important for event-driven multirate methods.

If the WR and event-driven multirate methods are used to perform a transient analysis of the unidirectional circuit in Fig. 5.1, the WR method would be more efficient. In general, for any unidirectional circuit the WR method would be more efficient than an event-driven method. This is because WR obtains an accurate solution for each node using a single waveform iteration whereas the event-driven approach has the overhead of scheduling events and rolling back the solution for step rejections. However, the circuit in Fig. 5.1 does not represent a realistic situation, since very few practical circuits are strictly unidirectional. In the inverter chain, there is usually a

nonlinear capacitor connecting nodes A and B and therefore WR would require a minimum of two iterations to compute the correct solution. As a result of coupling and feedback in real circuits, the partitioning algorithm and window size selection strategy become the important factors in determining the overall run time in WR. The event-driven approach can be viewed as a "windowless" multirate method and could potentially outperform WR when the coupling between the relaxation variables is large or when the windows are not selected optimally for the WR method. This is one reason why the event-driven approach is worth pursuing. Another reason stems from the fact that time moves incrementally forward in the event-driven approach and is well-suited for use in mixed-mode simulation programs (i.e., SPLICE, SAMSON, MOTIS).

5.3. PREVIOUS WORK IN EVENT-DRIVEN MULTIRATE METHODS

A number of previous event-driven multirate schemes are reviewed in this section. In particular, the methods proposed by Gear [Gea80], the Splice2 program [Kle84], the SAMSON program [Sak81] and the MOTIS3 program [Che84] are examined.

5.3.1. Gear's Methods

Three approaches proposed by Gear [Gea80, Wel81] are the "smallest-step-first", "recursive-divide-by-2" and "largest-step-first" algorithms. The "smallest-step-first" algorithm is the standard event-driven approach in which components are scheduled at the grid point and solved in order of increasing time. For the example given by Eqn. (5.1) with integration time-steps specified in Fig. 5.2, the order in which each of the variables are solved is illustrated in Fig. 5.3. The dormant model for each variable is based on extrapolation. The approach is specified more precisely in the following algorithm:

Algorithm 5.1 (Smallest-Step-First Multirate Method)

```
/* Backward-Euler Integration Assumed */
    SSF() {
        foreach ( i ∈ { 1, ··· ,n } ) {
            h_i ← PickStepSize(x_i(t_0) );
            Schedule(x_i , h_i );
        }
        t ← NextEventTime(0);
        while ( t ≤ T_stop ) {
            foreach ( event j at t ) {
                solve
                x_j(t) = x_j(t − h_j) + h_j f_j(x_1^D(t), x_2^D(t), ..., x_i^D(t), ··· , x_n^D(t));
                h_j ← PickStepSize(x_j(t));
                Schedule ( x_j , t + h_j );
            }
            t ← NextEventTime(t);
        }
    }
    ■
```

The routine *PickStepSize(x(t))* checks the accuracy of the solution $x(t)$ using a LTE

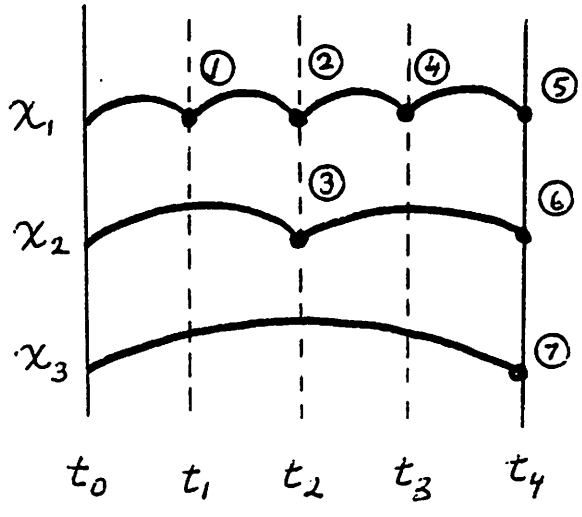


Figure 5.3 : Order of computation in the Smallest-step-first Algorithm

estimate and returns the next step size. The *Schedule(x,t)* routine schedules variable x at time t. The *NextEventTime()* routine identifies and returns the next time point after time t when one or more events are scheduled. The value $x_i^D(t)$ is obtained from the dormant model for x_i which is based on extrapolation.

As described previously, the main problem with the approach given in Algorithm 5.1 is due to step rejections. One strategy is to try to avoid step rejections by examining the reasons why they occur:

Observation 5.1: Step rejections in a given component, x_i, are primarily due to

a) changes in the function f_i that are not anticipated based on earlier values of x_i

or

b) changes in another variable x_j which couples into x_i through f_i.

■

Based on this observation, two techniques are used in [Gea80] to suppress rejections. The first is to perform a pre-integration of all the variables with their recommended steps, using extrapolations for other variables whenever necessary, to check if the recommended steps are reasonable. If the steps are acceptable, the integration proceeds in the usual manner. If not, the appropriate step sizes are reduced and checked again before applying Algorithm 5.1. The second modification is to check the effect of coupling between two variables, say x_i and x_j, on the error in the solution computed for x_i. The conjecture is that the variable which takes the larger step, x_i, may still end up rejecting its solution if the other variable, x_j, varies drastically from its extrapolated value. After each integration step for x_j, its computed value is checked against the original extrapolated value. If the effect of the predictor-corrector difference on x_i is large, then the step size of x_i is reduced. This involves calculating the Jacobian term

$\frac{\partial f_i}{\partial x_j}$ and, in general, involves computing a Jacobian term for each fanout component of x_j. The problem with this approach is the amount of additional overhead incurred in computing these Jacobian terms.

In another approach called "recursive-divide-by-2", all components in the system are integrated using the largest recommended step size, H. Those components which fail the accuracy criteria are re-integrated using one-half the previous step. This approach is applied recursively until the steps are selected without any rejections in any of the components. The method is specified in the algorithm below and illustrated in Fig. 5.4.

Algorithm 5.2 (Recursive-Divide-by-2 Multirate Method)

```
/* Backward-Euler Integration Assumed */

RDB2() {
    foreach ( i ∈ { 1, ··· ,n } ) {
        h_i ← PickStepSize(x_i(t_0));
        PutItemInList( i , ListA );
    }
    H ← max_{1≤i≤n} (h_i) ;
    i ← 0;
    repeat {
        i ← i + 1;
        ListB ← NULL;
        foreach ( i ∈ { 1, ··· ,2^i } ) {
          foreach ( event j in ListA ) {
            h_i ← H /2^i ;
            solve x_j(t +h )=x_j(t +h ) + h_j f_j(x^D(t +h ));
            if ( CheckAccuracy(x_j , h ) = FALSE )
              PutItemInList( j , ListB );
          }
        }
        ListA ← ListB ;
    } until ( ListA = NULL )
}
■
```

In this non-recursive version of the algorithm, two lists are maintained. *ListA* contains the components being integrated in the current iteration while *ListB* is

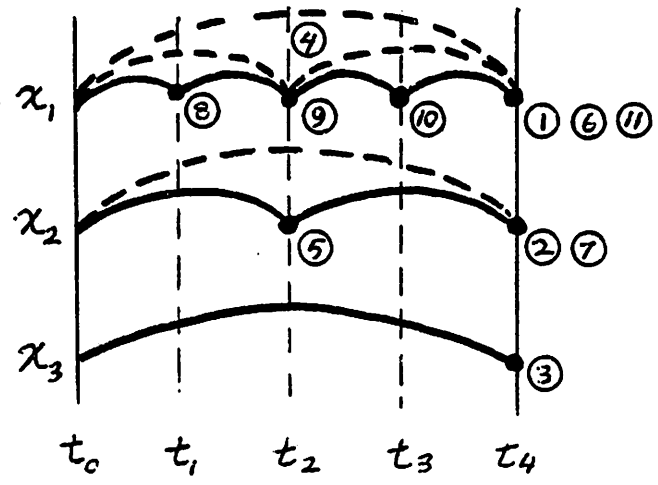


Figure 5.4 : Order of computation in the Recursive-divide-by-2 Algorithm

updated with the components to be integrated in the next iteration. The routine *PutItemInList(i, list)* places item i in a *list*. The *CheckAccuracy*(x_i, t) routine uses a LTE estimate to check the accuracy of the solution for x_i at time t.

The main advantage of this approach is that interpolation is used rather than extrapolation. That is, the dormant model for each component is the waveform from its previous iteration, as in WR. The problem with the method is that the components taking the larger steps may be affected by the faster components but this is not taken into account. However, the time-step control has some nice features and a modified version of it is used in conjunction with a new waveform-based relaxation method, as described in Chapter 6.

The "largest-step-first" method also has the property that interpolation can be used by some of the components in the system (but not all components as in "recursive-divide-by-2"). In this approach, the components with the largest step, H, are integrated first. If the solutions are rejected, the components are re-integrated using steps of $H/2$ along with any of the other components that use the half step size. This process continues until the variables with the smallest step sizes are solved. Then all components using the smallest steps are moved one step forward in time and solved again. The next step of the algorithm is to start with the components taking the next largest steps and work back to the components with the smallest steps. This process continues until the time point associated with the largest step H is reached. The order of the computation is illustrated in Fig. 5.5. This approach can be implemented as a recursive routine as done in Algorithm 5.3 below.

Algorithm 5.3 (Largest-Step-First Multirate Method)

```
/* Backward-Euler Integration Assumed */
 LSF() {
     foreach ( i ∈ { 1, ··· ,n } ) {
         h_i ← PickStepSize(x_i(t_0));
         Schedule(x_i , h_i );
     }
     H ← max_{1≤i≤n}(h_i) :
     h_min ← min_{1≤i≤n}(h_i) :
     RecursiveLSF(0,H );
     end;
 }

/* Recursive Largest-Step-First (LSF) routine */
RecursiveLSF(t ,h ) {
     foreach ( event j at t +h ) {
         solve x_j(t +h )=x_j(t ) + h_j f_j(x^D(t +h ));
         if (CheckAccuracy(x_j , t +h ) = FALSE) {
             h_new ← h /2 ;
             Schedule(x_i , t −h +h_new );
         }
         else {
             h_new ← PickStepSize(x_j(t +h ));
             Schedule(x_j , t +h +h_new );
```

```
        }
    }
    if(h = h_min) return;
    else {
        RecursiveLSF(t,h /2);
        t ← h /2;
        RecursiveLSF(t,h /2);
        return;
    }
} ■
```

This method suffers from problems similar to the recursive-divide-by-2 algorithm in that the slower components are integrated only once and, in effect, determine the solutions of the faster components. Note that since the components using the larger steps are integrated first, the faster components must be extrapolated in some way. A zeroth-order extrapolation is used to avoid potential stiff-stability problems [Gea80].

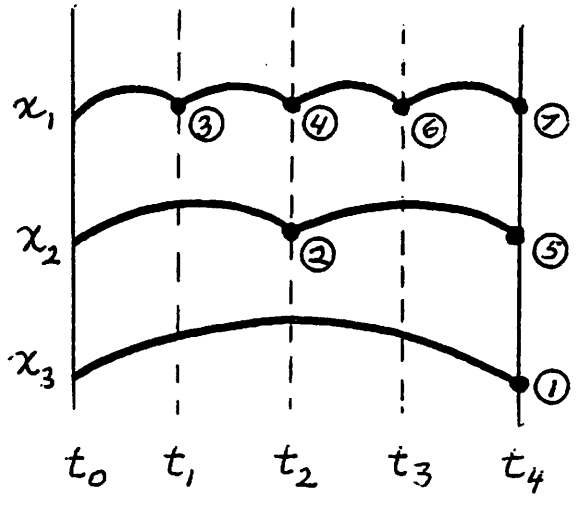


Figure 5.5 : Order of computation in the Largest-step-first Algorithm

One aspect not shown in Algorithm 5.3 is that the step size for a given component may be doubled if the LTE is very small. In practice, they are only doubled if the components are aligned with other components taking the larger step. Both Algorithms 5.2 and 5.3 use binary-weighted step sizes which are extremely useful for a number of reasons. This aspect will be described in more detail in a later section.

5.3.2. Circuit Simulators Using Event-driven Multirate Schemes

a. Time-step Control in SAMSON

The SAMSON program is an event-driven mixed circuit/logic simulation program for large integrated circuits [Sak81]. For electrical analysis, it uses a solution technique based on block LU decomposition techniques to decouple the system of equations, rather than using the standard approach or relaxation techniques. However this aspect is not germane to the discussion here. The relevant aspects of SAMSON are the time-step control algorithm, the dormancy model and the step rejection method in the multirate integration scheme. The integration method is based on the prediction based differentiation formulas due to Van Bokhoven [Van75] which are essentially a reformulation of the Backward-Difference Formulas (BDF) described in [Bra72]. The time-step control can be classified as a "smallest-step-first" method of Algorithm 5.1. The time-steps are chosen based on the following standard expression for the *a priori* LTE estimate:

$$E(t_{n+1}+h_{next}) = \frac{\prod_{i=1}^{k}(h_{next}+h_{i-1})}{\sum_{i=1}^{k}\frac{1}{(h_{next}+h_{i-1})}}\frac{\beta_k}{\alpha_k}E_{n+1} \qquad (5.2)$$

where $\beta_k = \sum_{j=1}^{k}\frac{1}{h_j}$, $\alpha_k = \prod_{j=1}^{k}h_j$, and k is the order of the integration method used. Eqn. 5.2 is used to compute the recommended step size h_{next}. A step is accepted if the a

posteriori LTE estimate, E_{n+1}, at time point t_{n+1} which is given by

$$E_{n+1} = \frac{x_{n+1}^{k+1} - \hat{x}_{n+1}}{h_{k+1}\beta_{k+1}}$$

satisfies the user specified error tolerance. In the above, $\hat{x}_{n+1}$ is the computed solution at t_{n+1} and x_{n+1} is a $k+1^{st}$ -order polynomial predictor evaluated at t_{n+1}. The dormant model in SAMSON is based on extrapolation. The original model[1] simply used a k th-order predictor with a $k+1$st-order corrector and the *a posteriori* error estimate as follows:

$$x^D(t_{n+1}+h) \approx x^{k+1}(t_{n+1}+h) - \frac{\beta_k}{\alpha_k} E_{n+1} \prod_{i=1}^{k+1} (h + h_{i-1})$$

Step rejections in SAMSON are handled using the same philosophy as in [Gea80] - that is, to try and avoid them. Two mechanisms are used to accomplish this. The first is use steps which are smaller than the size recommended by Eqn. 5.2, and this approach is commonly used in most other simulators. The second is to activate a dormant subsystem whenever one of its inputs deviates significantly from its expected behavior [Sak84]. However, when rejections do occur, SAMSON repeats the solution of only those components that reject their solutions. This simple scheme may introduce errors into other components.

b. Time-Step Control in MOTIS2

The multirate integration method in MOTIS2 [Che83] is based on iteration count time-step control. Initially, all components used the same time-step. Each component is solved and a new time-step is determined based on the number of iterations required to converge. If more than a certain number of iterations are required, the time-step is reduced by a factor of 2. If less than a certain number of iterations are required, the time-step is increased by a factor of 2. Some limits are placed on the largest and

[1] A more general dormant model has been proposed recently in [Sak85].

smallest allowed time-steps. While this approach is extremely efficient, it makes no explicit attempt to control the decoupling errors and integration errors.

c. Time-step Control in SPLICE2

The mixed-mode simulation program SPLICE2 [Kle84] also uses ITA to solve the circuit equations. A number of different time-step control algorithms have been implemented in this program including a multirate version of ITA. The details of the multirate approach are described in this section. The most interesting aspects of this approach are the windowing algorithm and back-up strategy. Before describing these two mechanisms, the following definition is necessary.

Definition 5.2 (Synchronization Points) : Any time point at which the solutions for two or more components are calculated is called a *local* synchronization point. Any point in time where the solutions to all components are performed is called a *global* synchronization point.

■

SPLICE2 uses windows as intervals over which the multirate method is applied. A window interval is divided into a number of subintervals of equal size and each associated time point within the window is called a grid point. These grid points are provided to encourage local synchronization since components can only be scheduled at grid points. Window edges are used as global synchronization points. The window size is based on three factors: the smallest recommended step of any component, H_{min}, the largest recommended step of any component, H_{max}, and the maximum number of grid points allowed in a window, MaxGridPoints. Usually H_{max} determines the window size if MaxGridPoints is large. Otherwise H_{min} determines the size. More precisely, H_{max} is compared to H_{min}×MaxGridPoints and the smaller of the two is used as the window

size. Input breakpoints[2] also act as global synchronization points and windows may be adjusted to align with them. The details of the window size selection is given in the following algorithm. The parameters associated with windows in SPLICE2 are given in Fig. 5.6(a).

Algorithm 5.4 : Window Size Selection in SPLICE2

```
/* All variables are global variables */
/* Time refers to the current simulation time */
 ComputeWindowSize() {
        WindowSize← MaxGridPoints×H_min;
        WindowSize← min(H_max,WindowSize);
        WindowEndTime← Time + WindowSize;
        WindowEndTime ← CheckBreakPoints(Time ,WindowEndTime);
        H_max← WindowEndTime - Time;
        H_min← H_max/MaxGridPoints;
 }
```

■

Once a window size is selected, components are scheduled at the appropriate grid points and solved using a smallest-step-first algorithm with dormant models based on extrapolation. When all components reach the end of the window, a global synchronization step is performed. A new window is selected and the simulation continues in this manner. The important steps in the algorithm are specified below.

Algorithm 5.5 (Multirate ITA in SPLICE2)

```
/* Backward-Euler Integration Assumed */
   Splice2ITA() {
        H_min← ∞;
        H_max← 0;
        foreach ( i ∈ { 1, ··· ,n } ) {
            h_i ← PickStepSize(x_i (t_0) );
            H_min← min ( h_i ,H_min );
            H_max← max ( h_i ,H_max );
        }
        Time ← NextEventTime(0);
        while ( Time ≤ T_stop ) {
            ComputeWindowSize();
            foreach ( i ∈ { 1, ··· ,n } ) {
                h_i ← min(t +h_i , WindowEndTime);
                Schedule(x_i , h_i );
```

[2] Breakpoints are transition points, associated with external inputs, from one value to another. They usually cause the simulator to use smaller time-steps than would otherwise be necessary.

```
        }
        while ( Time ≤ WindowEndTime ) {
            foreach ( event j at Time ) {
                t ← Time;
                solve
                x_j(t)=x_j(t − h_j) + h_j f_j(x_1^D(t), x_2^D(t), ..., x_i^D(t), ..., x_n^D(t));
                if ( CheckAccuracy(x_j, h) = TRUE) {
                    h_j ← PickStepSize(x_j(t));
                    h_j ← min(t + h_j, WindowEndTime);
                    Schedule ( x_j, t + h_j );
                }
                else { /* Step Rejection */
                    h_j ← PickStepSize(x_j(t));
                    RejectSolution(x_j, h_j );
                }
            }
            Time ← NextEventTime(Time);
        }
    }
}
■
```

When a step rejection occurs in computing a solution for a component, x_i, a new step size, h_{new}, is calculated. The action taken in rejection mode depends on the value of h_{new} compared to $H_{\min}$ (i.e., the smallest allowed step in the current window). If h_{new} is larger than $H_{\min}$, the component and its fanout nodes are backed-up to the point of the rejection, $t_{i,n}$, and scheduled at the new time point, $t_{i,n} + h_{new}$, and the simulator continues normally. However, if h_{new} is less than $H_{\min}$ some problems arise. A new grid size, and hence a new window size, is required to accommodate the smaller step. Since there may still be a number of components scheduled at future time points in the window, while other components have already computed solutions at earlier times using the old grid size, a complicated procedure is necessary to make them consistent with the new window based on h_{new}. Instead, the SPLICE2 program forces a global synchronization point, starts a new window and then continues normally. This is illustrated in Fig. 5.6(b) and described in the algorithm below.

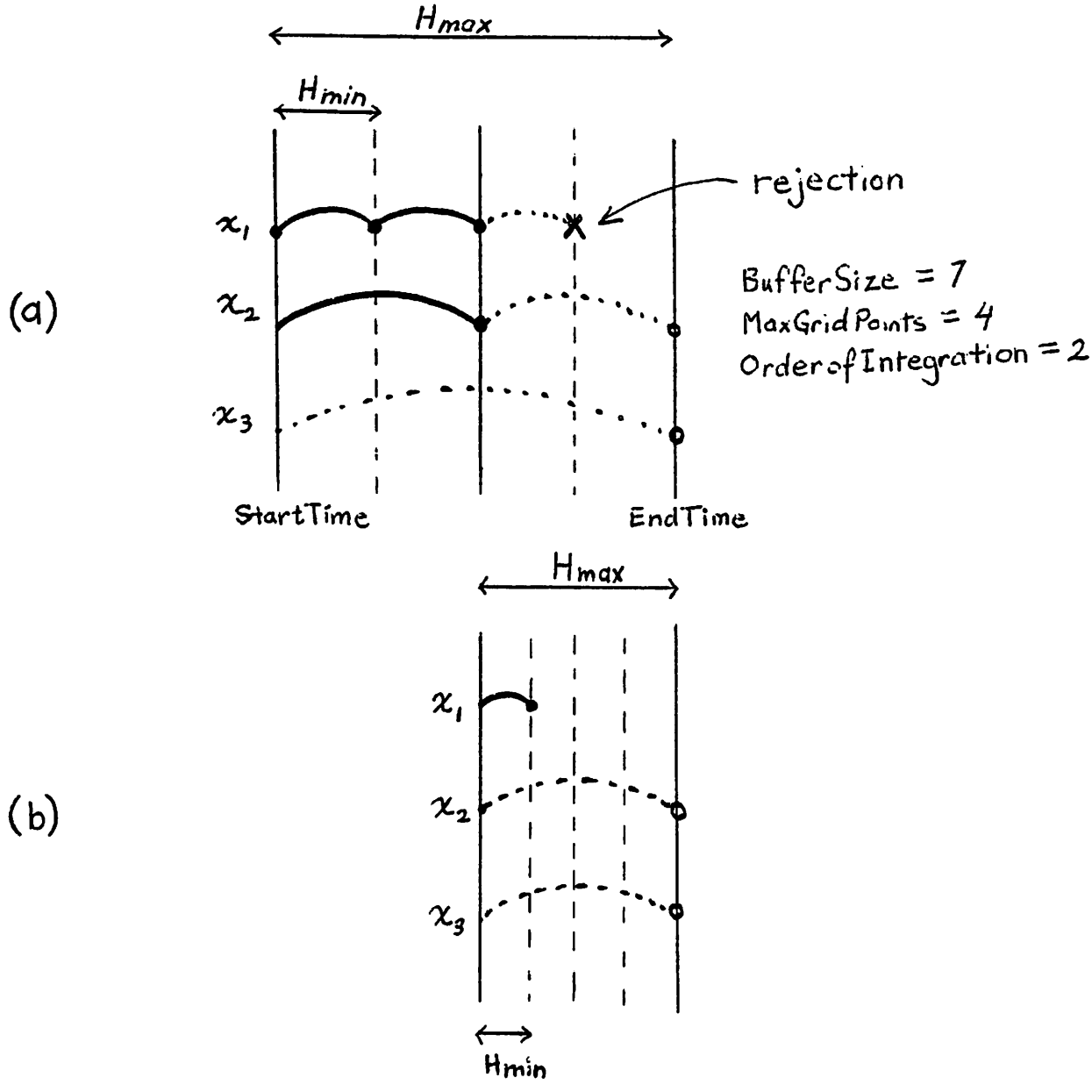


Figure 5.6 : (a) Window parameters (b) New Window after Rejection

Algorithm 5.6 (Step Rejection in SPLICE2)

```
RejectSolution(x_i , h_new ) {
    if ( h_new ≥ H_min ) {
        BackUpSolution(x_i );
        h_new ← H_min × Trunc(h_new /H_min );
        Schedule(x_i , t_i,n + h_new );
        foreach ( x_j ∈ Fanouts( x_i ) ) {
            BackUpSolution(x_j );
            Schedule(x_j , t_j,n + h_new );
        }
    }
    else {
        H_min ← h_new ;
        ComputeWindow();
        ForceGlobalSync();
    }
}
```

The *BackUpSolution(x)* routine simply discards all computed solutions back to the point of the rejection. The *Trunc(y)* function returns the closest integer that less than the real number y.

The multirate scheme in SPLICE2 has a number of limitations. One source of problems is the windowing mechanism. As indicated in Fig. 5.6(a), "BufferSize" is a window parameter which is related to "MaxGridPoints" as follows:

MaxGridPoints = BufferSize - (Order of Integration) - 1

If BufferSize is chosen to be a small value, the window size is MaxGridPoints × H_{min}. Unfortunately, components using steps larger than this window size will be constrained to take a step equal to the window size, and components using steps slightly smaller than the window size will be required to take two steps equal to half the window size. The strategy for choosing BufferSize does not permit efficient exploitation of multirate behavior. One may considering choosing a larger value for BufferSize in which case the window size would be equal to H_{max}. However this leads to a slightly different problem: the faster components are able to compute many more points since the grid spacing is H_{max}/MaxGridPoints but if a rejection occurs for a slower component, the resulting

roll-back could be very costly! This is, in fact, what the results in Table 5.1 indicate.

Statistic	BufferSize=25	BufferSize=5
MaxGridPoints	23	3
Run Time	2128sec	1341sec
Order of Integration	1	1
No. of Equations Solved	270100	179415
No. of Accepted solutions	100290	62866
Total no. of iterations	269452	178612
No. of 'useful' iterations	183566	147553
'Useful'/Total iterations	0.68	0.82
No. of Windows Used	284	351
No. of Rejected solutions	414	172
No. of Forced Global Sync	175	172

Table 5.1 Effect of Buffer Size Variation on CRAMB circuit

Although only two BufferSize values are shown, the results are quite representative of the nature of this time-step control. The results indicate that only 68% of the computed solutions were accepted when BufferSize=25, whereas 82% were accepted with BufferSize=5. This is due to the impact of the solution roll-back in the two cases. As a result, the program runs faster with the smaller BufferSize even though it requires many more windows (351 compared to 284).

The other notable point is that every rejection in the BufferSize=5 case causes global synchronization. Since the grid spacing is based on H_{min} the probability is high that a rejection will result in global synchronization. For the BufferSize=25 case, less than 50% of the rejections forced a global synchronization. However the total number was still larger than the BufferSize=5 case. Based on these results, and others reported in [Kle84], it is apparent that a backup strategy requiring global synchronization is not the right approach.

One obvious way to improve the SPLICE2 approach is to try and avoid global synchronization altogether. Global synchronization is useful for one practical reason - to reduce the memory requirements due to the history which must be maintained in case

of roll-back. A global synchronization point provides a point beyond which the program never needs to roll back. An additional benefit of global synchronization is that the solution of all variables are checked against each other for mutual consistency and accuracy. In a virtual memory environment, the waveform storage overhead is not a major problem. In a physical-memory environment it may be important as it would limit the size of problem that could be simulated. However, in SPLICE2, global synchronization is used frequently: at window edges, at breakpoints for any input source and, most importantly, as part of step rejection. This has a significant impact on the performance. To avoid this, the window size should be based only on $H_{min}\times$ BufferSize and BufferSize should be made as large as possible. If regridding is required, a simple solution would be to keep the solutions that have been obtained with the current grid size and re-schedule all pending events on a new grid. If any waveform storage buffers overflow, a global synchronization point can be defined. Although this operation seems rather involved, event scheduling overhead represents such a small portion of the run time compared to function evaluation that it is not a major point of concern.

5.4. A NEW MULTIRATE SCHEME FOR ITA

The multirate integration methods suggested by Gear have one shortcoming in that they assume little or no coupling between the fast and slow components. This assumption is not entirely valid in the circuit simulation problem. In fact, in Gear's approaches, there is no attempt to correct errors in the fast components due to errors in the initial integration of the slow components in Algorithms 5.2 and 5.3. The event-driven approach of Algorithm 5.1 is ruled out by Gear due to the roll-back problem encountered. Based on the SPLICE2 results, the concern over the cost of roll-back is well-founded.

The objective here is to overcome the efficiency and accuracy limitations of previous event-driven multirate techniques. A new approach is proposed which retains the inherent advantages of ITA but uses different step sizes to solve different components. An appropriate name for it might be "all-steps-together" since the components are integrated over one step in a cooperative manner, even though each component may be using different step sizes. The approach uses dormant models based on interpolation, rather than extrapolation as used in most other event-driven simulators. Each component is integrated twice over the same step: once as a trial integration and a second time as a final integration. A trial integration is defined as one which uses a mix of interpolated and extrapolated values from other components. Trial integrations by the slower components provide solutions which can be interpolated by the faster components. A final integration is defined as one which only uses interpolated values from the other components and is used to re-compute and check the solution once the other components have "caught up" to the component being integrated. For the fastest components, only a final integration is performed since all other components can be interpolated.

The step rejection control is the other feature which distinguishes this method from previous attempts. In the previous approaches, there was some resistance to rolling back the solutions for step rejections due to its potential cost. However, due to interactions between the slow and fast components in circuit simulation, the event-driven approach must include the ability to roll the solutions back to an arbitrary time point to guarantee a reasonable level of accuracy. In the approach taken here, there is a deliberate attempt to control errors in this manner without incurring a significant time penalty.

5.4.1. The Basic Technique

Consider the system of nonlinear differential equations specified by Eqn. (5.1). The recommended step sizes are shown in Fig. 5.2. In the new scheme, the solutions for each component are approximated using one Newton iteration in the sequence x_1, x_2, x_3, even though they are being integrated to different points in time. This constitutes the first relaxation iteration. In the second relaxation iteration, the process is repeated in the same sequence and it continues in this manner until all components converge. Note that during the iterative process component x_1 uses interpolated values from x_2 and x_3 to determine its solution and is therefore a *final* integration. However, variables x_2 and x_3 use a mix of interpolated and extrapolated values and so these are only *trial* integrations.

After completing the iterative process, each component selects a new step size and is scheduled as shown in Fig. 5.7. The next sequence of iterations is defined by the bold arcs. That is, variable x_2 re-integrates its first step, while x_1 performs the integration for its second step. These are, by definition, final integrations for both x_1 and x_2. Again, only one Newton iteration is performed on each component before moving to the next component, but the iterative sequence is carried to convergence. Once convergence is obtained, new step sizes are computed for each component. The simulation continues in this manner until t_4 is reached. Then, all components perform final integrations of their last step to synchronize their solutions. This algorithm cannot be classified as largest-step-first or smallest-step-first since the processing order depends on the rank of each component and the solutions are obtained by iterating across a "ragged" boundary in time.

The multirate scheme can be implemented by using two time point edges which define an "integration time slot". If a component takes a step which begins at the first

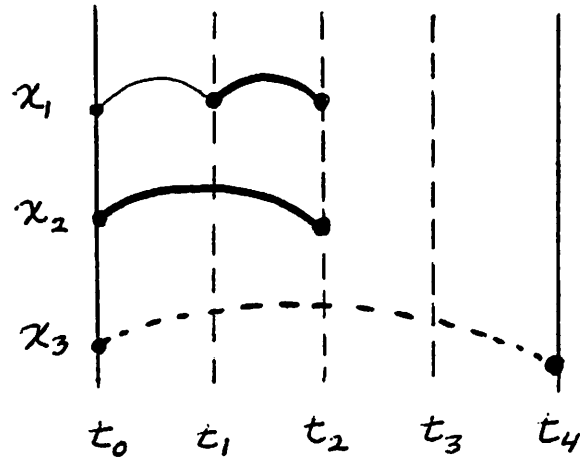


Figure 5.7 : Second Sequence of Iterations in Multirate ITA

time point edge, it performs a trial integration. However, if its endpoint is equal to the second time point it performs a final integration step. This is depicted in Fig. 5.8. Components x_1 and x_4 perform trial integrations, component x_2 performs a final integration and component x_3 is not evaluated. The details of the algorithm are given below:

Algorithm 5.7 (Multirate Iterated Timing Analysis)

```
/* Backward-Euler Integration Assumed */

MultirateITA() {
    timePoint1 = 0;
    timePoint2 = 0;
    while ( timePoint2 ≤ T_stop ) {
        foreach ( i ∈ { 1,...,n } ) {
            h_i ← PickStepSize(x_i(t_i,old));
            t_i,old ← t_i,new ;
            t_i,new ← t_i,old + h_i ;
        }
        allConverged ← TRUE;
```

```
        timePoint1 ← timePoint2;
        timePoint2 ← min_{1≤i≤n} t_{i,new};
        repeat {
            foreach ( i ∈ { 1...n } ) {
                if ( (t_{i,old} = timePoint1) or (t_{i,new} = timePoint2) ) {

                    compute
                    x_i^{k+1}(t_{i,new}) ← x_i^k(t_{i,new}) + F_i(x_i^k(t_{i,new}))/F'_i(x_i^k(t_{i,new}))
                      where F_i(x_i) = x_i(t_{i,new}) − x_i(t_{i,old}) − h_i f_i(x^D(t_{i,new}));

                    if ( |x_i^{k+1} − x_i^k| > ε_1 or |F_i(x_i^{k+1})| > ε_2 )
                            allConverged = FALSE;
                }
            }
        } until (allConverged = TRUE)
    }
}
■
```

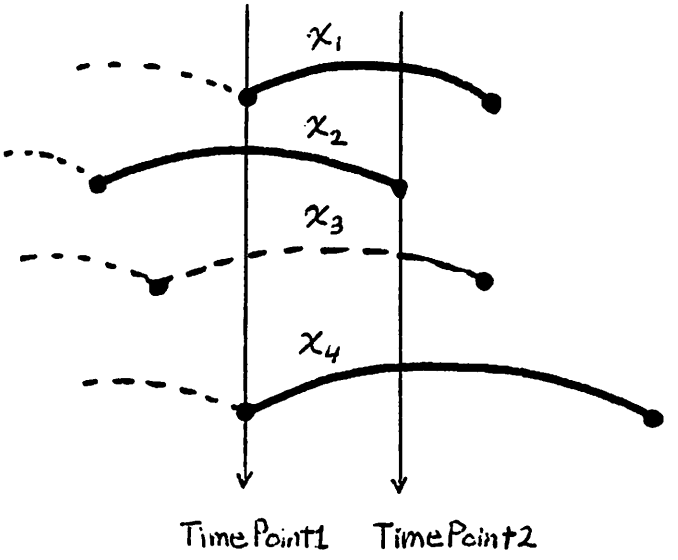


Figure 5.8 : Time Slot defining the active components

5.4.2. Refinements to the Basic Technique

If two variables take approximately the same time-step, then one will be forced to use extrapolation, and the other interpolation, during the relaxation process. This may be expensive depending on the order of the extrapolation/interpolation method, but more importantly, it may introduce errors in the computation. One way to reduce this decoupling error is to encourage more synchronization. This can be accomplished by using binary weighted step sizes. The idea is to allow two variables, which are taking approximately the same step, to take *exactly* the same step so that they may be solved together. It is not enough to simply use binary-weighted step sizes to insure that these synchronization points occur. In fact, "leap-frogging" of variables using the same step size may still occur. To avoid this problem, it is necessary to enforce a rule that time-steps may not be doubled unless the component is aligned with a grid point which corresponds to the desired step size [Gea80].

The event scheduling mechanism for the new approach is based on the same principle as used in the SPLICE1.7 and SPLICE2 programs. However there are one or two minor differences. If one component schedules its fanouts for processing, the fanouts merely check to see if any roll-back operation is necessary but are not forced to perform any unnecessary time-step cutting or re-evaluation. In this multirate scheme, the latency condition can be viewed as the use of a zeroth-order integration method (see Chapter 4) to pick step sizes. That is, the components are *always scheduled* at some point in the multirate scheme, but if they satisfy the latency condition much larger steps are permitted.

5.4.3. Selective-Backup Strategy

Step rejections must be handled properly in the event-driven environment to avoid unacceptable levels of error in the solutions. Therefore, all solutions between

two global synchronization points must be saved since, in principle, rejections can cause the solutions to be rolled back to the last global synchronization point. The memory requirements of such an undertaking may be substantial but it is an important requirement to guarantee accurate solutions. Perhaps more importantly, the rejection and roll-back procedures must be efficient or they may dominate the run time.

When a step rejection occurs in the SPICE2 program [Nag75], all solutions are rejected and the entire system is re-evaluated using a smaller step. A similar technique is used in SPLICE2 [Kle84] if regridding is necessary within a window. This effectively propagates the rejection to all parts of the circuit and is a rather pessimistic view of the effect of a rejection. On the other hand, SAMSON re-integrates only those components which reject their step, but the new solutions are not propagated to the other components [Sak83]. While this approach is efficient, it is far too optimistic and does not directly control the error. The technique proposed in this section explicitly controls errors due to a rejection, but attempts to limit the temporal and spatial domain of these rejections.

Intuitively, a step rejection at a particular component, x_i, in the system should not necessarily affect every component in the system. Rather, it should only affect a subset of the components limited to its local spatial domain defined below.

Definition 5.3 (Spatial Domain of a Step Rejection):

The spatial domain, $D_s(x_i)$, of a given component x_i is the set of all components $\{ x_j \}$ which are sensitive to changes in x_i, either directly or indirectly through another component $x_k \in D_s(x_i)$. This domain may vary in size depending on the dynamic coupling between the components of the system at any given time. ■

Ideally, a *selective backup* strategy should be used such that only the components in the

spatial domain, $D_s(x_i)$, are backed-up and re-integrated whenever a rejection occurs at a particular node. For example, if a rejection occurs at an internal node, x_i, in a long chain of inverters, the rejection will affect the solution of nodes in the neighborhood of x_i. This may include, for instance, x_{i-2}, x_{i-1}, x_{i+1} and x_{i+2}. Only these nodes would be rolled back and re-integrated over the appropriate interval of time.

A conservative backup strategy would require that all components $x_j \in D_s(x_i)$ be re-integrated in the rejection interval $[t_{start}, t_{end}]$. A more efficient approach would be to note the time point, t_{diff}, in $x_i^R(t), t \in [t_{start}, t_{end}]$, obtained in the roll-back step (hence the superscript R), which deviates far from the initial integration solution, $x_i^I(t)$ (hence the superscript I), which other components used during their integration process. This reduced interval is defined as the temporal domain of the rejection:

Definition 5.4 : (Temporal Domain of a Step Rejection)

Assume that component x_i performs a re-integration of its solution in the interval $[t_{start}, t_{end}]$ due to a step rejection. The temporal domain of a rejection, $D_T(x_i)=[t_{diff}, t_{end}]$, is the associated subinterval for which the new solution affects at least one neighboring component of x_i. ■

The components in $D_s(x_i)$ are only required to perform a backup operation in the subinterval $[t_{diff}, t_{end}]$ rather the possibly larger interval $[t_{start}, t_{end}]$. This process is illustrated in Fig. 5.9. The solution is computed for waveform, $x_i^I(t)$, at time points t_0, t_2, and t_4. The solution is computed at all time points for waveform $x_i^R(t)$. These are used as sampling points to determine the point of the rejection which should propagate to other components. Since the new solution is far from the old solution at sampling point t_3, the fanouts of x_i are scheduled to reject their solution back to at least that point in time. This is specified in the algorithm below.

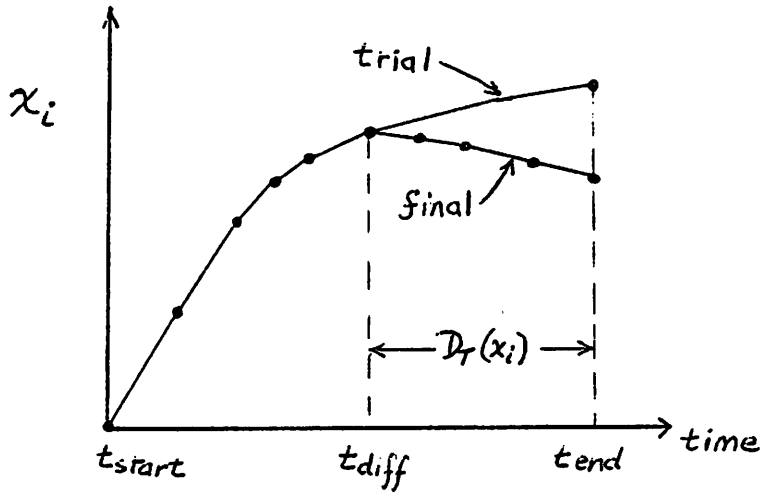


Figure 5.9 : Rejection Control

Algorithm 5.8 (Step Rejection)

```
/* t_{i,n} is the n^{th} time point for x_i */
/* t_{i,n+1} is the n+1^{st} time point for x_i */

if(CheckAccuracy(x_i , h_i ) = FALSE) {  /* step rejection */
      if ( FINAL_INTEGRATION_STEP ) {
            /* save old waveform */
            x_i^L ← WaveSetUpPrevious(x_i , t_{i,n} );
            /* generate new waveform using smaller steps */
            x_i^R ← WaveReComputeSoln(x_i , t_{i,n} , t_{i,n+1});
            /* determine the point at which waveforms differ */ diffTime ←
            WaveCompare(x_i^L ,x_i^R ,t_{i,n} ) ;
            ScheduleFanouts( x_i , diffTime);
      }
      else {  /* Initial integation step */
            h_i ← h_i /2;  /* just cut the time-step */
            ScheduleFanouts( x_i , t_{i,n} );
            /* ensure that fanouts are updated */
      }
} ■
```

The *WaveSetUpPrevious(x,t)* routine saves a copy of the waveform associated with

variable x starting at time t. The *WaveReComputeSoln*(x, t_1, t_2) routine produces a new solution for x in the interval $[t_1, t_2]$ using a number of smaller step sizes. The *WaveCompare*(x_1, x_2, t) routine compares the waveforms x_1 and x_2 starting at point t to determine the time point at which the two differ.

The fanout components perform rejections in the same manner. That is, they re-integrate their solution in the rejection interval and determine at which point the previous waveform differs significantly from the new waveform and then schedule components in their spatial domain to reject their solutions back to that point. This is specified in more detail in Algorithm 5.9.

Algorithm 5.9 (Solution Roll-Back)

```
if (t_{i,schedTime} < timePoint1 ) {
      x_i^{old} ← WaveSetUpPrevious(x_j , t_{i,schedTime} );
       /* generate new waveform using smaller steps */
      x_i^{new} ← WaveReComputeSoln(x_i , t_{i,n} , t_{i,n+1});
       /* determine the point at which waveforms differ */
      diffTime ← WaveCompare(x_i^{old}, x_i^{new}, t_{i,n} );
      ScheduleFanouts( x_i , diffTime);
} ■
```

At some spatial distance from the original offending component, the previous and new waveforms of the components being backed-up will not differ, and this implicitly defines its spatial domain since the temporal domain is reduced to zero.

5.4.4. Summary

The multirate ITA scheme presented above is a "windowless" multirate integration scheme. The solution is obtained by iterating across a "ragged" boundary in time. Each step is integrated twice, once for a trial integration to move a component ahead of the others, and a second time for a final solution when the other components have caught up. The main advantage of this approach is that is uses interpolation to obtain values of neighboring components when computing the final solution of a particular

component. Previous schemes ignored the effect of a rejection on neighboring nodes, or propagated the rejection to all components in the system. The approach taken here is to extend the notion of event-driven, selective-trace so that only those components that are affected by the rejection are backed-up. Each time a rejection occurs, the offending component compares its previous waveform in the rejection interval with the new waveform and schedules all fanout components to be re-integrated in the interval where a significant difference exists.

The diagrams in Fig. 5.10 show the sequence of integration steps. The dotted lines indicate the iteration boundary at each phase of the computation. Note that when a rejection occurs, the roll-back mechanism propagates the error to components affected by the rejection and not to all components in the system. This selective backup strategy is key factor in determining the overall efficiency of the integration process.

5.5. INCREMENTAL REPARTITIONING

As mentioned earlier, static partitioning does not allow full multirate integration of the internal variables of a subcircuit. Dynamic partitioning as implemented in RealAx [Mar85] has a similar problem over the period of a simulation window. In reality, a circuit experiences incremental changes in the coupling as the simulation evolves. In order to follow the circuit behavior, repartitioning should be performed on an incremental basis, whenever the coupling changes in the circuit. The ITA approach allows this kind of *incremental repartitioning* to be performed. Of course, it would be far too expensive to repartition the whole circuit at each time point. Instead, only those subcircuits which are good candidates for repartitioning are processed. The candidates for repartitioning can be determined using heuristics.

In the incremental partitioning technique, two processes may take place. In the first process, single subcircuits may be repartitioned into smaller subcircuits to increase

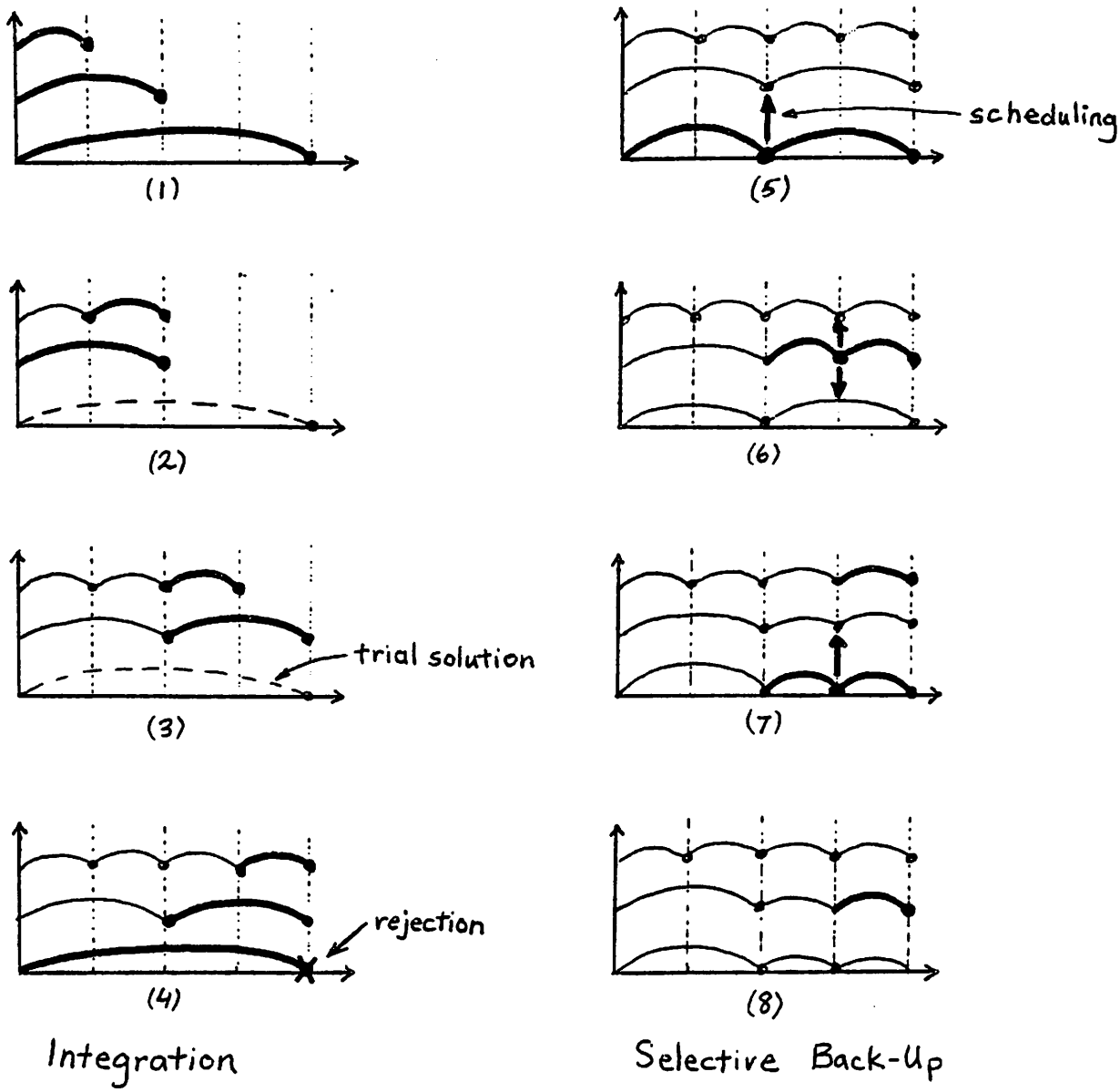


Figure 5.10 : Solution process in the Multirate ITA Scheme

efficiency of the multirate integration scheme. In the second process, two or more subcircuits may be merged to form a larger subcircuit to improve the convergence of the iterative process. These two processes should only occur when certain factors indicate that it would be worthwhile to perform the associated operations. In theory, the exact coupling between the nodes in a subcircuit could be computed as a basis for the decision but this would be very expensive. Therefore, heuristic measures are necessary to determine whether a subcircuit should be partitioned into smaller subcircuits or a pair of subcircuits should be merged.

One way to decide if a subcircuit should be divided into smaller subcircuits is to keep a count of the number of active devices which have changed their region of operation such that the coupling between two or more nodes has been modified. If this number is large, relative to the number of nodes in the subcircuit, the subcircuit should be repartitioned. Another crude, but useful, measure of coupling between two variables is the ratio of their recommended step sizes. If the recommended time-steps for the nodes within a subcircuit vary greatly, it is an indication that the multirate property is not being exploited efficiently, since the smallest time-step is used to compute the solution for the subcircuit. Therefore, a useful heuristic can be developed by examining time-steps. If all the time-steps for the nodes in a subcircuit are approximately the same, the subcircuit is not considered for repartitioning. However, if the time-steps are quite different, then the partitioning function is activated. The *actual* repartitioning is performed using the capacitive and resistive coupling information and not by using the time-steps. The heuristics given above are intended to provide strong indicators that repartitioning should be performed. It does not imply that the subcircuit will be modified. That would depend on the exact values of coupling in the circuit during the repartitioning phase.

The second problem is to decide if two or more subcircuits should be merged. Good candidates for merging can determined by examining certain relationships between subcircuits. For example, if two neighboring subcircuits use the same step size, then it is worthwhile to check if any of the nodes in the two subcircuits are tightly-coupled. If two nodes in different subcircuits are tightly-coupled, then a merge operation should be performed. Another indication that two neighboring subcircuits should be merged is if both require many relaxation iterations to obtain a solution at a particular time point.

There are three phases in this incremental repartitioning strategy. The first step is the "filtering" operation to identify the subcircuits that should be merged and the subcircuits that need to be repartitioned. This operation would be extremely efficient due to the simplicity of the heuristics being used. The second phase is to actually perform the repartitioning operation. In this phase, all subcircuits which pass through the filter are processed. It is important that this operation be reasonably fast or it may offset the improvements in convergence and multirate integration efficiency. To assist in repartitioning, information about the conductances and capacitances could be stored in the device data structures during the simulation. These values can be used during the repartitioning phase rather than having to regenerating them every time. Also, a flag indicating the region of operation of the transistors could be used to quickly decide what the node groupings should be. This would be useful for bipolar circuits. Finally, the notion of *hard* and *soft* links should be used. That is, some links between two nodes should never be broken during the course of the simulation (for example, two nodes connected by a very small resistor). These are referred to as hard links. The remaining connections are referred to as soft links. Repartitioning should only be performed across soft links to minimize the total time required by the operation. The third and final phase is to rebuild the data structures and create new matrices to reflect the new subcircuit definitions. Most of the data structures should be node-oriented

wherever possible to minimize the number of changes required when new subcircuits are defined. Since most of the new partitions are incremental changes to the old partitions, certain functions (such as subcircuit merging, etc.) should be made to run as fast as possible.

One alternative which may reduce the overhead of dynamic partitioning is to keep the data structures of previous partitions available for re-use at a later time. In this approach, the original data structure for any subcircuit which is divided into two or more smaller subcircuits is saved. Later, if the smaller subcircuits are merged together, the data structure for the original subcircuit is re-used and the data structures for the smaller subcircuits are saved temporarily. The rationale behind this approach is that often digital circuits feature a cyclic operating behavior. Clock signals tend to connect and disconnect two portions of a circuit in a repetitive manner. The above strategy attempts to exploit this property to reduce the overhead of dynamic partitioning.

A preliminary version of the incremental partitioning algorithm was implemented in the SPLICE3.1 program [Ma85] and showed a 5-10% improvement in runtime for a few MOS circuit examples. The program used the iteration count heuristic to perform merge operation. It is anticipated that larger speed improvements would be obtained for bipolar circuits.

5.6. SIMULATION RESULTS USING MULTIRATE ITA

The multirate scheme for ITA described in this chapter has been implemented in the SPLICE3.2 program. The results from simulations performed using the program are presented in Table 5.1. The first column contains the circuit name. The second column contains the total number of iterations needed to produce the solution. In the next three columns, the number of iterations used in the trial, final and rollback phases, respectively, are provided. The numbers in the table indicate that approximately 60% of the

iterations were used in the initial integrations, 30% in the final integration and 10% in the rollback procedure. This implies that it is not sufficient to simply perform the trial integration step, as done in the smallest-step first algorithm. The final integration and rejection control iterations are necessary to improve accuracy of the solution. The fact that the selective backup scheme requires only 10% of the time is an indication that it is working in an efficient manner.

The results shown in Table 5.2 compare the SPLICE3.2 to the RELAX2.3 program, which uses Waveform Relaxation. Both methods are also compared to the maximum speed-up that can be obtained by exploiting multirate behavior as determined in Chapter 3. The SPLICE3.2 program is faster than RELAX2.3 for most of the circuits provided in table. In these cases, RELAX2.3 was not able to choose the window sizes properly resulting in an excessive number of iterations. In the case of the CRAMB circuit, the unidirectionality of this circuit allows WR to outperform the multirate ITA method, as expected. For the EPROM circuit, the RELAX2.3 program was able to choose proper window sizes for the problem resulting in the performance improvement over SPLICE3.2.

Circuit	Total Iters	Trial Iters	Final Iters	Rollback Iters
DECPLA	44237	24965	15573	3699
CKT3	518298	316654	145736	55908
SCDAC	616261	367163	193281	55817
CRAMB	242731	142469	73034	27228
EPROM	2749764	1795281	645557	308926

Table 5.1: Simulation results using SPLICE3.2

Circuit	Size (nodes)	Ideal Speed-up	Actual Speed-up (SPLICE3.2)	Actual Speed-up RELAX2.3
uP Control	56	4.6	1.1	1.0
CRAMB	149	27.9	4.0	9.5
SCDAC	154	8.5	1.4	0.7
CKT3	312	25.9	2.1	1.0
EPROM	630	63.9	3.1	5.4

Table 5.2: Ideal Speed-up compared to Actual Speed-up

If the actual speed-up of SPLICE3.2 and RELAX2.3 are compared to the ideal values, it is clear that both methods fall far short of the ideal case. The reasons why the programs do not reach the maximum speed-up values are similar to the ones presented in Chapter 4. These factors are:

- a conservative time-step control
- static partitioning
- scheduling overhead (SPLICE3.2)
- window size selection (RELAX3.2)

In short, many more time-points are calculated in practice and each solution point is more expensive than the direct approach. It should be noted that both programs outperform SPICE2 on all examples. In addition, these examples are small enough so that model evaluation dominates the runtime. For very large circuits, the relaxation-based programs are expected to be much faster than direct methods since the linear equation solution time will be the dominant factor in the direct methods.

5.7. CONCLUSIONS

In summary, a number of basic differences between the event-driven multirate methods and Waveform Relaxation were described in this chapter. The main difference is that the event-driven approach obtains the solution incrementally in time whereas WR uses a waveform-based approach. Other differences are due to the nature of the time-domain decoupling error and the impact of step rejections on the two methods. Previous implementations of event-driven schemes were described along with a number of circuit simulators that use event-driven methods. Then the new multirate scheme based on ITA was described. In this new scheme, the solution is obtained by iterating across a ragged boundary in time. Each step is integrated twice, once for a trial integration to move a component ahead of the others, and a second time for a final solution when the other components have caught up. The main advantage of this approach is that is uses interpolation to compute the values of neighboring components when computing the solution of a particular component. The rejection control scheme selectively backs-up only those components affected by a step rejection of a particular component. This limits the propagation of the rejection in both time and space and improves the efficiency and accuracy of the event-driven scheme compared to previous methods. The results presented in this chapter indicate that this approach is more efficient than the Waveform Relaxation method on a number of examples.

CHAPTER 6

A MULTIRATE INTEGRATION METHOD USING WAVEFORM-NEWTON

6.1. INTRODUCTION

The nonlinear relaxation methods described in the previous chapters were used to solve the circuit equations using single, common time-steps to exploit circuit latency. These techniques were adapted to exploit the multirate property of circuits by using different time-steps to solve different components in the system. While the performance of the event-driven multirate integration approach was much better than the version which exploited only latency, a somewhat complicated rejection control mechanism was necessary to damp the effect of step rejections whenever they occurred. Furthermore, each component used interpolated values from trial solutions of neighboring components when computing its own final solution. Therefore, a time-domain decoupling error was introduced into the final solutions, the exact nature of which is not well understood.

In this chapter, the nonlinear relaxation methods are extended to function spaces to exploit the multirate property of circuits using a new waveform-based relaxation algorithm. The algorithm uses the Waveform-Newton approach [Kan59] and combines the advantages of Waveform Relaxation and Iterated Timing Analysis for the solution of moderately coupled multirate systems. The method has been applied to circuit simulation in the work of Van Bokhoven [Bok83] and Palusinski [Gua83]. The main differences in their approaches and the approach taken here are due to the equation formulation, time-step control and other refinements to the basic relaxation algorithm to obtain large speed improvements over other existing simulators.

The motivation for the Waveform-Newton method is given in Section 6.2. The method is derived and applied to the circuit simulation problem in Section 6.3. Its use in conjunction with the Waveform Relaxation algorithm is given in Section 6.4 along with an iterative stepsize refinement strategy which improves the accuracy of the numerical integration as the relaxation iterations approach convergence. Simulation results are also presented in Section 6.4 and conclusions are provided in Section 6.5.

6.2. MOTIVATION FOR A NEW APPROACH

Both the ITA and WR algorithms attempt to exploit the loose or unidirectional coupling of MOS digital circuits, in that the relaxation converges rapidly when applied to loosely-coupled systems. However, circuits contain elements that introduce tight coupling between two or more nodes. In order to improve the convergence speed, relaxation-based simulators perform a partitioning step to group tightly-coupled nodes together into subcircuits [Whi83]. Direct methods are used to solve each subcircuit and a relaxation method, whether WR or ITA, is applied at the subcircuit block-level rather than to individual equations. However, even the remaining coupling between subcircuits affects the convergence speed and this coupling may vary with time during the simulation. Therefore, many relaxation iterations may still be required to achieve convergence.

The advantage of the WR algorithm is that it inherently exploits multirate behavior by solving the differential equations in a decoupled fashion. However, each iteration is potentially expensive as it involves solving a set of nonlinear differential equations accurately. For ITA, each iteration is much cheaper, involving only a single Newton iteration, but the exploitation of multirate behavior is more complicated, as seen in the previous chapter. What is desired is an approach which combines the advan-

tages of these two relaxation methods. One way to accomplish this is to use the single Newton iteration strategy of the ITA method in the context of the WR method. That is, rather than solving the iteration equations of WR accurately in the inner loop, use an approach which approximates the solution in much the same way that a single Newton iteration approximates the solution for each nonlinear equation in ITA. A direct extension of the Newton method to function spaces would permit this kind of approach, and this method is referred to here as Waveform-Newton[1]. The details of the Waveform-Newton method are described in the following section.

6.3. WAVEFORM-NEWTON (WN)

6.3.1. The Space of Continuous Functions

In order to derive the Waveform-Newton method, a brief review of the space of continuous functions is presented. The main purpose of this section is to show the relationship between Euclidean n-space and function spaces. Recall that Euclidean n-space, $\mathbb{R}^n$, consists of all ordered n-tuples given by $x=(x_1,x_2,x_3,\ldots,x_n)$, where x defines a distinct point and the components of x are real numbers. Within this n-space, two operations are possible: vector-vector addition and scalar-vector multiplication. Euclidean n-space with these two operations defined is called a vector space of dimension n. The length, or *norm*, of a vector x is a mapping $\|\bullet\|$ from $\mathbb{R}^n$ to $\mathbb{R}^1$ which satisfies the following properties [Var62]:

Properties of the Vector Norm:

$$\begin{array}{ll} (i) & \|x\| \geq 0 \\ (ii) & \|x\| = 0 \text{ if, and only if, } x = 0 \\ (iii) & \|\alpha x\| = |\alpha|\,\|x\|\,,\ \alpha \in \mathbb{R} \\ (iv) & \|x+y\| \leq \|x\| + \|y\|\,,\ y \in \mathbb{R}^n \end{array} \qquad (6.1)$$

[1] The descriptive term "Waveform-Newton" is used here rather than Newton-Kantorovich to associate the method with Waveform Relaxation and because of other minor modifications. However, the basic conver-

Any vector space which satisfies these properties is called a *normed space*.

An analogous situation exists in function spaces [Mar74]. For example, consider the set of functions, V, with elements $f : A \rightarrow \mathbb{R}^m$. The operations of addition and multiplication are defined in a similar way to the Euclidean n-space. That is,

$$(f_1+f_2)(x)=f_1(x)+f_2(x), \qquad f_1, f_2 \in V$$

$$(\lambda f_1)(x)=\lambda(f_1(x)), \quad \lambda \in \mathbb{R}$$

Therefore, V is a vector space consistent with the earlier definition. In addition, if C_b is defined as all functions $f \in V$ such that f is bounded and continuous, then C_b is also a vector space since the sum of two continuous functions is continuous and $\alpha f(x) \in C_b$ if $f(x) \in C_b$. The measure of the size of f, or the norm of f, is given by

$$\| f \| = sup \{ |f(x)| \mid x \in A \}, \quad f \in C_b$$

which is known to exist since f is bounded. This norm satisfies properties analogous to those given in (6.1) with the appropriate modifications to denote functions. As a result, the space C_b can be viewed in the same way as $\mathbb{R}^n$ except that each element in C_b is a *function* rather than a *point* as in $\mathbb{R}^n$. C_b is only one of the many possible spaces of functions.

The advantage of considering C_b in the same way as $\mathbb{R}^n$ is that, in many cases, the properties, concepts, theorems and proofs for $\mathbb{R}^n$ can be carried over to function spaces in a straight-forward manner. As an example, consider the definition of a Cauchy sequence in $\mathbb{R}^n$ [Mar74]:

Definition 6.1 A sequence $x_k \in \mathbb{R}^n$ is called a Cauchy sequence if, for every $\epsilon > 0$, there is an N such that $l, k \geq N$ implies $\| x_k - x_l \| < \epsilon$. ■

---

gence proof of this method is due to Kantorovich.

Theorem: A sequence x_k in $\mathbb{R}^n$ converges to a point in $\mathbb{R}^n$ if, and only if, it is a Cauchy sequence. ■

The above definition and theorem provide an important test for convergence in $\mathbb{R}^n$ since the Cauchy condition does not involve the limit point explicitly. An equivalent condition exists for function spaces. That is, a sequence of functions, f_k, is called a Cauchy sequence if, for any $\epsilon>0$, there is an N such that $k, l \geq N$ implies that $\| f_k - f_l \| < \epsilon$. An important concept in connection with convergence in functions spaces is that of uniform convergence [Mar74].

Definition 6.2 Let $f_k : A \rightarrow \mathbb{R}^m$ be a sequence of functions with the property that for every $\epsilon>0$ there is an N such that $k \geq N$ implies $| f_k(x) - f(x) | < \epsilon$ for all $x \in A$. Under these conditions, it is said that f_k converges uniformly to f and this property is usually written as $f_k \rightarrow f$ (uniformly). ■

Another way to state Definition 6.2 is that the maximum absolute difference between f_k and f decreases as $k \rightarrow \infty$. This condition is strong enough to guarantee that if a Cauchy sequence of continuous functions f_k converges, the limiting function f will also be continuous.

This brief introduction to function spaces has been provided as a background for the derivation of the Waveform-Newton method. In the sections to follow, the Newton method and the nonlinear relaxation methods are extended from algebraic or Euclidean spaces to function spaces and are shown to have practical application to the circuit simulation problem. Theorems stating the conditions for which the methods are guaranteed to converge uniformly are also presented.

6.3.2. Derivation of the Waveform-Newton Method

Consider the following single nonlinear differential equation:

$$\dot{x}(t) = -f(x(t)), \quad x(0)=X, \quad t \in [0,T] \tag{6.2}$$

In the standard approach, the equation is solved using a stiffly-stable implicit integration method. At each time point, a nonlinear algebraic equation is solved using a damped Newton-Raphson method and, to insure stability, the iteration is carried to convergence.

Next, consider solving Eqn. (6.2) by reversing the order of the two numerical techniques. That is, apply Newton's method to the differential equation before applying an integration method. Eqn. (6.2) can be formulated as a nonlinear problem by rearranging:

$$F(x(t)) = \dot{x}(t) + f(x(t)) = 0. \tag{6.3}$$

The application of Newton's method to this problem produces

$$F'(x^k(t))\left[x^{k+1}(t) - x^k(t)\right] = -F(x^k(t)) \tag{6.4}$$

This equation should be viewed as a function space extension of the algebraic Newton method — sometimes referred to as functional linearization. Note that the equation has the same form as the usual Newton method but each term in the expression is a function of time, i.e. a waveform in the window interval [0,T]. The first term, $F'(x^k(t))$, can be obtained by extending the definition of a derivative. Recall that a function, $g(x)$, is considered differentiable, at some $x \in (a,b)$, if the limit

$$g'(x) = \lim_{\Delta \to 0} \frac{g(x+\Delta) - g(x)}{\Delta} \tag{6.5}$$

exists. This expression can be rewritten as

$$\lim_{\Delta \to 0} \frac{|g(x+\Delta) - g(x) - g'(x)\Delta|}{|\Delta|} = 0 \tag{6.6}$$

Using these expressions, the definition given below follows naturally [Mar74]:

Definition 6.3:

The mapping $F:D \subset \mathbb{R}^n \rightarrow \mathbb{R}^m$ is *Frechet-differentiable* at $x \in$ *interior* (D) if there exists a linear operator $F'(x):\mathbb{R}^n \rightarrow \mathbb{R}^m$ such that

$$\lim_{\|\Delta\| \rightarrow 0} \frac{\|F(x+\Delta) - F(x) - F'(x)\Delta\|}{\|\Delta\|} = 0 \tag{6.7}$$

■

Using a function space equivalent to Definition 6.3, the term $F'(x(t))$ of Eqn. (6.4) can be obtained, as derived below. The dependence of the terms on t is dropped for notational convenience. Using Eqn. (6.3) and expanding $F(x+\Delta)$ as a Taylor series:

$$F(x+\Delta) = \frac{\partial}{\partial t}(x + 1\cdot\Delta + 0\cdot\Delta^2 + \cdots) + f(x) + \frac{\partial f(x)}{\partial x}\Delta + \cdots$$

and retaining only the first two terms of the expansion, the following expression is obtained:

$$F(x+\Delta) - F(x) \approx \dot{\Delta} + \frac{\partial f(x)}{\partial x}\Delta$$

This expression can be used in Eqn. (6.7). After applying the limiting operation, one obtains

$$F'(x)\Delta = \dot{\Delta} + \frac{\partial f(x)}{\partial x}\Delta \tag{6.8}$$

Next, combining Eqns. (6.8) and (6.4), and letting $\Delta = x^{k+1} - x^k$, one obtains

$$(\dot{x}^{k+1} - \dot{x}^k) + \frac{\partial f(x^k)}{\partial x}(x^{k+1} - x^k) = -\dot{x}^k - f(x^k)$$

and, rearranging slightly, the WN algorithm is obtained:

$$\dot{x}^{k+1} = -\frac{\partial f(x^k)}{\partial x}x^{k+1} + \frac{\partial f(x^k)}{\partial x}x^k - f(x^k). \tag{6.9}$$

Note that this is a linear differential equation in x^{k+1} with time-varying coefficients. Here, x^k as an "input" since it is a known waveform. This time-varying

linear problem can be solved by applying an integration method and selecting a number of discretization time points. Since the problem is linear at each time point, it can be solved in one step. For example, if the trapezoidal method is used to solve the WN algorithm in Eqn. (6.9), the following expression is obtained:

$$[\frac{2}{h}+\frac{\partial f\,(x_{n+1}^{k})}{\partial x}](x_{n+1}^{k+1}-x_{n+1}^{k}) = -\frac{2}{h}[x_{n+1}^{k}-x_{n}^{k}]$$

$$-f\,(x_{n+1}^{k})-f\,(x_{n}^{k})-[\frac{2}{h}-\frac{\partial f\,(x_{n}^{k})}{\partial x}](x_{n}^{k+1}-x_{n}^{k}) \qquad (6.10)$$

Once the waveform, $x^{k+1}(t)$, is computed from Eqn. (6.10), the next iteration is performed using (6.9). This iterative process continues until convergence is achieved. Hence, the Waveform-Newton method converts the problem of solving the nonlinear differential equation in Eqn. (6.2) to the problem of solving a sequence of time-varying linear differential equations given by Eqn. (6.9).

It is interesting to compare Eqn. (6.10) to the expression obtained using the standard method to solve Eqn. (6.2) as given below:

$$[\frac{2}{h}+\frac{\partial f\,(x_{n+1}^{k})}{\partial x}](x_{n+1}^{k+1}-x_{n+1}^{k}) = \qquad (6.11)$$

$$-\frac{2}{h}[x_{n+1}^{k}-x_{n}]-f\,(x_{n+1}^{k})-f\,(x_{n})$$

Note that an additional term appears on the right-hand-side (RHS) of Eqn. (6.10). To illustrate the source of this extra term, consider the sequence of waveforms in Fig. 6.1. In WN, the linearization process at each point is done using the values from the previous iteration. For example, waveform x^{k} is generated by linearizing around the waveform x^{k-1}, and likewise, the waveform x^{k+1} is generated by linearizing around the waveform x^{k}.

In computing the value $x^{k+1}(t_4)$, the term $[\frac{2}{h}-\frac{\partial f\,(x^{k}(t_3))}{\partial x}](x^{k+1}(t_3)-x^{k}(t_3))$ appears in the RHS vector. This term is a result of the fact that the correct solution has

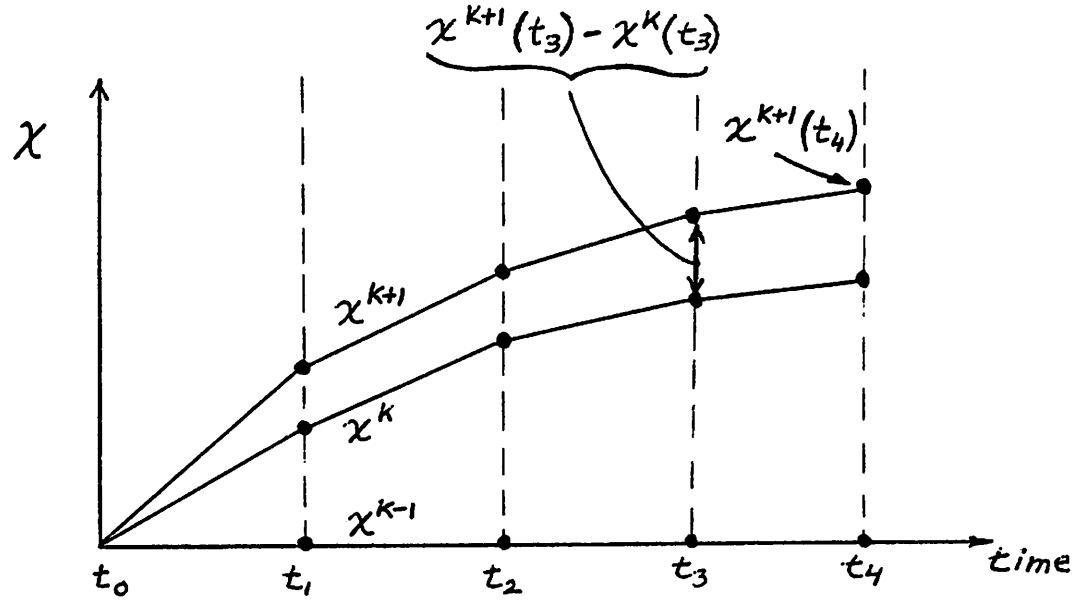


Figure 6.1 Waveform-Newton Iterations

not been obtained at the previous time point, t_3. In the standard approach given by Eqn. (6.11), the extra term is not present since the Newton-Raphson iteration is carried to convergence at time t_3 before moving to the next time point, t_4.

From the above example, it may seem that using a single Newton-Raphson iteration at each time point in Eqn. (6.11) is equivalent to a single iteration of the WN algorithm in Eqn. (6.10). However, this is not the case. The first method obtains an approximate solution to the nonlinear differential equation (using a method which is similar to timing analysis), while the second approach obtains an accurate solution to a related time-varying linear problem.

6.3.3. Application to Circuit Simulation

In the previous section, the WN method was introduced using a simple one-dimensional nonlinear differential equation. In this section, the method is applied to a *system of equations* provided by the circuit simulation problem. The same sequence of steps are performed here as in the one-dimensional case. The starting point of the derivation is the charge formulation:

$$\dot{q}(v(t)) = -f(v(t),u(t)), \quad v(0)=V, \quad t \in [0,T]. \tag{6.12}$$

where $v(t) \in \mathbb{R}^n$ and $u(t)$ is the set of input sources. By rearranging this system of equations, it can be formulated as a nonlinear problem:

$$F(v(t)) = \dot{q}(v(t)) + f(v(t),u(t)) = 0. \tag{6.13}$$

To solve this nonlinear problem, the Waveform-Newton method is used:

$$J_F(v^k(t))(v^{k+1}(t) - v^k(t)) = -F(v^k(t)) \tag{6.14}$$

Note that the iteration variables, $v^{k+1}(t)$ and $v^k(t)$, and the right-hand-side term, $-F(v^k(t))$, are all vectors of waveforms. The first term, $J_F(v(t))$, is analogous to the Jacobian matrix except that, in this case, it is a matrix-valued function of time. That is, each element of the matrix is a waveform which spans the period of interest, [0,T], as illustrated in Fig. 6.2 for a system of two equations and two unknowns. This term can be obtained by using the Frechet derivative given in Eqn. (6.7). Using Taylor series expansions for $f(v+\Delta)$ and $q(v+\Delta)$:

$$F(v+\Delta v) = \frac{\partial[q(v+\Delta v)]}{\partial t} + f(v+\Delta v) \tag{6.15a}$$

$$\frac{\partial[q(v+\Delta)]}{\partial t} = \frac{\partial}{\partial t}[q(v)+\frac{\partial q}{\partial v}\Delta + \frac{\partial^2 q}{\partial v^2}\Delta^2 + ...] \tag{6.15b}$$

$$f(v+\Delta) = [f(v)+\frac{\partial f}{\partial v}\Delta + \frac{\partial^2 f}{\partial v^2}\Delta^2 + ...] \tag{6.15c}$$

Then

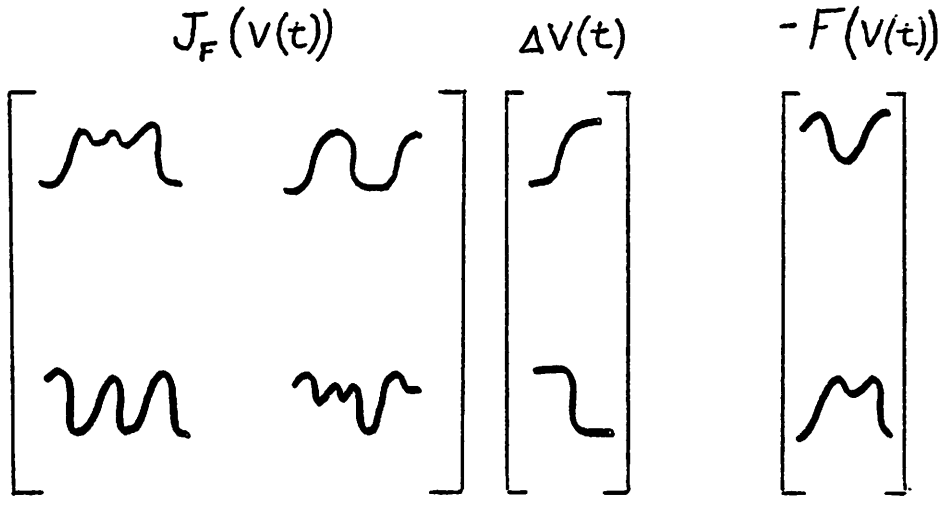


Figure 6.2 Waveform-Newton Equations

$$F(v+\Delta) - F(v) = \frac{\partial}{\partial t}\left[\frac{\partial q}{\partial v}\Delta + \frac{\partial^2 q}{\partial v^2}\Delta^2 + \ldots\right] \tag{6.16}$$

$$+ \left[\frac{\partial f}{\partial v}\Delta + \frac{\partial^2 f}{\partial v^2}\Delta^2 + \ldots\right]$$

and applying the limiting operation specified in Eqn. (6.7)

$$J_F(v)\Delta = \frac{\partial}{\partial t}\left[\frac{\partial q}{\partial v}\Delta\right] + \left[\frac{\partial f}{\partial v}\Delta\right] \tag{6.17}$$

If Eqn. (6.17) is applied to Eqn. (6.14), with $\Delta = v^{k+1} - v^k$, the following equation is obtained:

$$\frac{\partial}{\partial t}\left[\frac{\partial q(v^k)}{\partial v}\Delta\right] + \frac{\partial f(v^k, u)}{\partial v}\Delta = -\frac{\partial q(v^k)}{\partial t} - f(v^k, u) \tag{6.18}$$

and finally rearranging the expression, the Waveform-Newton algorithm is obtained:

$$\frac{\partial}{\partial t}\left[\frac{\partial q(v^k)}{\partial v}\Delta + q(v^k)\right] = -f(v^k, u) - \frac{\partial f(v^k, u)}{\partial v}\Delta \qquad (6.19)$$

This equation is used to compute $v^{k+1}(t)$, which is the set of waveforms at the $k+1$st iteration. The new waveforms are applied to (6.19) to compute $v^{k+2}(t)$ and the iterative process continues in this manner. A natural question to ask is whether or not the WN method converges to the solution and, if so, under what conditions? The following theorem has been proved [Whi85c]:

Theorem 6.1: For any system of the form $\dot{q}(v) = -f(v, u)$ in which $\frac{\partial q(v)}{\partial v}$ is Lipschitz continuous with respect to v for all u, and f is continuously differentiable, the sequence generated by the WN algorithm converges uniformly to the correct solution *for any initial guess.* ■

Remark 6.1: The above theorem differs from the algebraic Newton method which requires that the initial guess to be close enough to the correct solution to guarantee convergence [OrRh70]. For WN, any initial guess can be used and the method is guaranteed to converge to the correct solution. Of course, the initial guess waveforms must satisfy the the initial conditions at time t=0. ■

Remark 6.2: WN is similar to the algebraic Newton method in that the rate of convergence is quadratic. ■

6.3.4. Waveform-Newton Algorithm

The Waveform-Newton algorithm to solve systems of the form of Eqn. (6.12) is given below.

Algorithm 6.1 (Waveform-Newton Algorithm)

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

$$\frac{\partial}{\partial t}\left[q(v^k(t)) + \frac{\partial q(v^k(t))}{\partial v}(v^{k+1}(t) - v^k(t)) \right] -$$
$$f(v^k(t)) + \frac{\partial f(v^k(t))}{\partial v}(v^{k+1}(t) - v^k(t)) = 0$$

  for ($v^{k+1}(t)$; $t \in [0,T]$).
  if ($\max_{t \in [0,T]} \| v^{k+1}(t) - v^k(t) \| \leq \epsilon$) convergence = TRUE;
  $k \leftarrow k+1$;

} **until** (convergence)
■

In the first step of this algorithm the set of guess waveforms, $v^0(t)$, are specified such that the initial conditions (at t =0) are satisfied. Note that the equations in the inner loop are the original differential equations linearized by the WN method. These equations are solved by standard integration methods to find the waveform at iteration $k+1$. That is, in the $k+1^{st}$ iteration, the waveforms, $v^{k+1}(t)$, $t \in [0,T]$, are generated by linearizing around the previous waveforms, $v^k(t)$, and solving these equations using a multistep integration method. This process is repeated until convergence is obtained. Note that relaxation is not used in this WN algorithm. It uses direct matrix techniques to solve the system at each time point in the window interval $[0,T]$.

While it is possible to use the above WN algorithm to solve the entire system of differential equations, there are a number of reasons why this is not a recommended approach. First, it would suffer from the same problem as the direct methods, namely that it would not be well-suited to the simulation of large problems since the linear equation solution time would dominate the overall runtime. Second, it would be difficult to exploit latency and multirate behavior.

One may wonder whether the WN method would be more efficient than the standard direct methods. Of course, for linear problems the standard circuit simulation approach and the WN method are identical. For nonlinear problems, it is useful to compare the two approaches in terms of the window size used. The "window" size in the standard approach is equal to the step size while in WN it is usually much larger. As a result of the smaller window sizes, the standard approach converges much more rapidly. It would not be appropriate to use WN as a general simulation approach because the nonlinearity of circuit and the window size would determine its performance. In fact, in the best case, WN would only equal the performance of the standard approach, assuming that the step sizes used are the same in both cases.

6.4. WAVEFORM RELAXATION-NEWTON (WRN)

Rather than using WN as a stand-alone simulation technique, it is more effective to use this algorithm in conjunction with the WR algorithm [Lel81,Bok83,Pal83]. In presenting the precise algorithm for this Waveform Relaxation-Newton algorithm (WRN) the following notation will be used.

$$v^{k,i}(t) = (v_1^k(t), \ldots, v_{i-1}^k(t), v_i^{k-1}(t), \ldots, v_n^{k-1}(t))^T .$$

Algorithm 6.2 (Gauss-Seidel WRN Algorithm)

$k \leftarrow 0$;
convergence = FALSE;
guess waveform $v^0(t) : t \in [0,T]$ such that $v^0(0) = v_0$;
repeat {
  foreach ($i \in \{ 1,\ldots,n \}$) {

    solve

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

    for ($v_i^k(t) : t \in [0,T]$), with the initial condition $v_i^k(0) = v_{i_0}$;
  }
  if ($\max_{1 \leq i \leq n} \max_{t \in [0,T]} \| v_i^k(t) - v_i^{k-1}(t) \| \leq \epsilon$) convergence = TRUE;
  $k \leftarrow k+1$;

}
until (convergence)
■ .

The **repeat** and **foreach** constructs, taken together, represent the WR loop, which is the outer loop. Each differential equation is solved in the inner loop. In Algorithm 6.2, the WN method is used to solve each differential equation. Note that this algorithm can be viewed as a function space extension of the relaxation-Newton algorithms used in ITA. Therefore, the same strategy is used in Algorithm 6.2 as in ITA to reduce the cost of each iteration. That is, instead of using a number of WN iterations in the inner loop, the solution of each nonlinear differential equation is approximated using one step of the Waveform-Newton algorithm. This results in a one-step WRN algorithm and is analogous to the one-step Gauss-Seidel-Newton method used in ITA.

Remark 6.3: The convergence criterion in Algorithm 6.2 is not sufficient to guarantee an accurate solutions since it only checks that the waveforms are close enough. The original KCL condition specified in Eqn. (6.13) must also be satisfied. This KCL condition can be checked using the strategy described in Chapter 2 for the relaxation-Newton methods. ■

Algorithm 6.2 has been implemented in a new program called SPLAX (derived from the names SPLICE and RELAX) [Whi85b]. In the sections to follow, the details of the implementation are described.

6.4.1. An Efficient Time-Step Control for WRN

It is possible to make the WRN algorithm even more efficient using a time-step control which is well-suited to the nature of the converging waveforms. Since the waveforms computed initially are far from the correct solution it is not worth spending much time computing them accurately. One way to reduce the amount of computation

on the early iterations is to use large steps initially, and then refine the step sizes as the iterations progress. Using this approach, most of the work is performed when the waveforms are close to the correct solution.

The largest possible step that can be taken on the first iteration is one that is equal to the window size. The window size would have to be much smaller than the one used in standard WR for this approach to work well[2] and, at present, the window size is some multiple of the user requested plot increment. When a step of this size is taken it produces the waveform shown in Fig. 6.3(a). On the second iteration, the window interval is divided in half and two time-steps are taken, as shown in Fig. 6.3(b). On each successive iteration, additional timepoints are only added if the LTE criterion at a particular time point is not satisfied. The appropriate interval is then divided in half on the *next* iteration. For example, if, on the second iteration, the LTE is too large at time point T/2 but acceptable at time point T, then only the first subinterval is divided in half *on the third iteration*. The size of the second subinterval is not modified. Therefore, three discretization time points are used in the third iteration. This is shown in Fig. 6.3(c).

Remark 6.4: Note that this approach is similar to Algorithm 5.2 due to Gear [Gea80] except that all subsystems are solved on every iteration and the subintervals are divided only if the LTE is not acceptable. ■

This time-step control strategy has the advantage that, in general, time-steps will be placed more efficiently to control truncation error than if the standard predicted error criteria is used. This is because the time-step selection is based on the more accu-

[2] Especially if the devices are highly nonlinear as in the case of the diode in the forward-bias region of operation, as described in a later section.

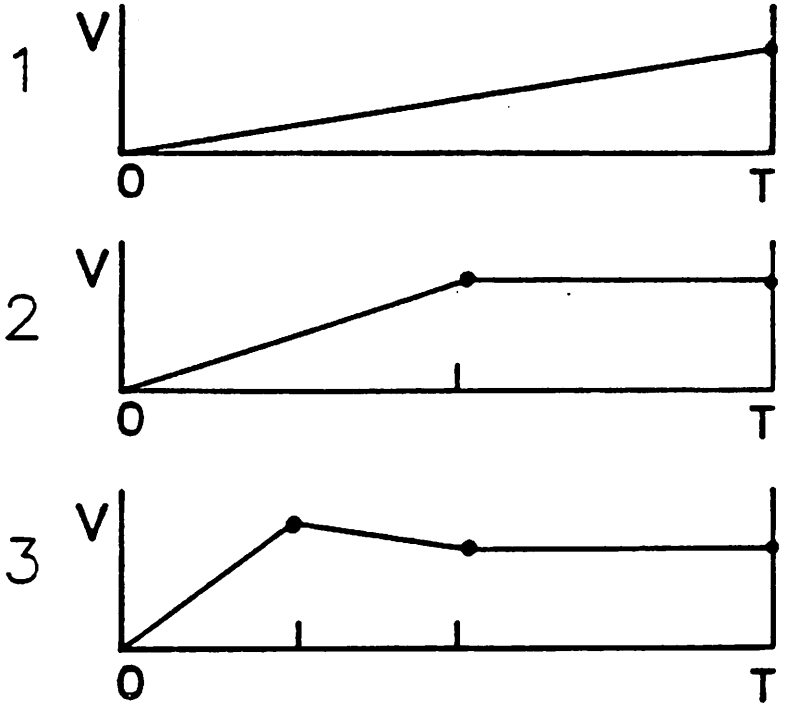


Figure 6.3 WRN Time-Step Control

rate *a posteriori* error estimates available from the previous iteration. However, there are situations where too many time points may be placed in a region due to a "wave-front" moving through the window as the iterations progress. One way to avoid this problem is to remove time points on subsequent iterations if the LTE is exceptionally small at a time point. There may be problems of "thrashing", that is, time points which are added on one iteration are removed on the next iteration and then added again on the third iteration. To avoid this, it may be necessary to remove time points only if the waveform is latent and the LTE is small in the interval of interest. This particular optimization has not been implemented in the current version of SPLAX.

6.4.2. Choice of Integration Method

The trapezoidal method for solving $\dot{x}(t) = f(x(t))$ is given by

$$x_{n+1} = x_n + \frac{h}{2}[f(x_{n+1}) + f(x_n)]$$

and is the most accurate A-stable multistep integration method [Dah63]. For this reason, it is commonly used in a number of circuit simulators such as SPICE, SPLICE and RELAX. However, due to the nature of the time-step control described above, the trapezoidal method may not be the best choice for WRN. If large steps are used, the trapezoidal method may produce solutions which oscillate in time, referred to as "point-to-point ringing" (see [Nag75]). To understand this, consider solving the test problem:

$$\dot{x}(t) = -\lambda x(t) \tag{6.20}$$

If the trapezoidal integration method is used, the following equation is obtained:

$$x_{n+1} = x_n - \frac{h}{2}(\lambda x_{n+1} + \lambda x_n)$$

In this case, the equation can be rearranged to give x_{n+1} in terms of x_n as follows:

$$x_{n+1} = \frac{\left|1 - \frac{h\lambda}{2}\right|}{\left|1 + \frac{h\lambda}{2}\right|} x_n .$$

Notice that if $|h\lambda| \geq 2$, the method produces a solution which oscillates as a function of time. The solution for the case of $|h\lambda| = 3$ is given in Fig. 6.4.

Normally this oscillatory behavior is harmless since it can be controlled by using tight tolerances in the time-step selection scheme, and because the oscillations usually die quickly. However the time-step strategy for WRN uses large time-steps during the initial iterations. As a result, the trapezoidal method produces this oscillatory characteristic and it requires many extra iterations to remove this anomalous behavior. This unnecessarily increases the overall run time for the method.

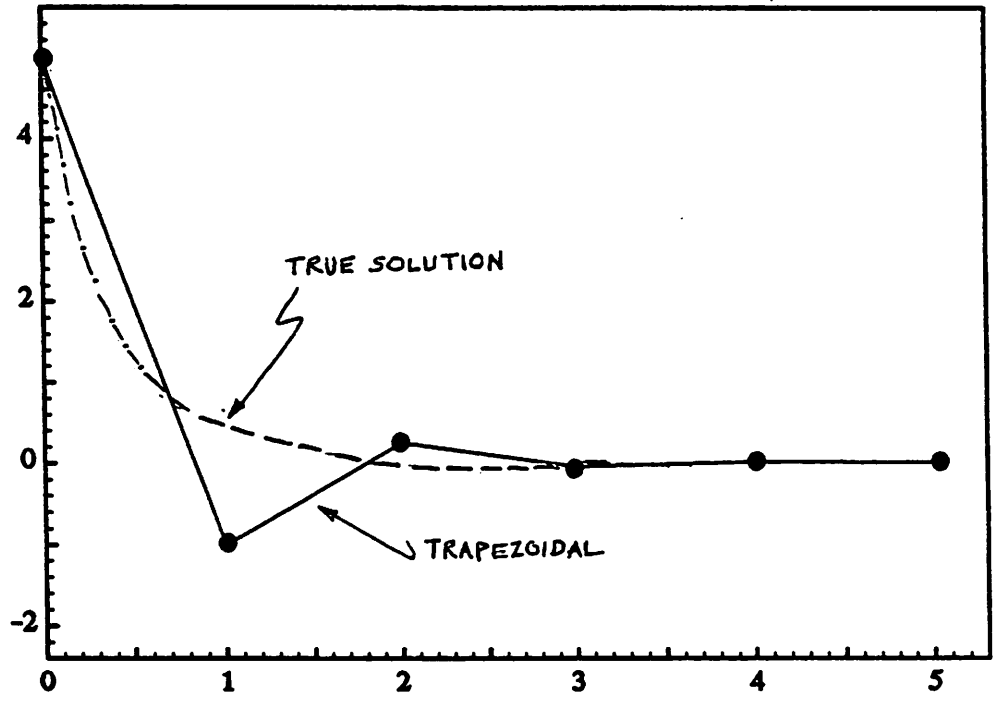


Figure 6.4 Trapezoidal Ringing
"$x_0 = 5.0$ $|h\lambda| = 3$"

To avoid this problem, the more stable second-order Gear method [Gea71] given by:

$$x_{n+1}+\alpha_1 x_n+\alpha_2 x_{n-1}+\alpha_3 f(x_{n+1})=0$$

is used in SPLAX. The coefficients, α_1, α_2, and α_3, are functions of the time-steps. For the variable time-step case:

$$\alpha_1 = -\frac{(h_{n-1}+h_n)^2}{h_n(2h_{n-1}+h_n)}$$

$$\alpha_2 = \frac{(h_{n-1})^2}{h_n(2h_{n-1}+h_n)}$$

$$\alpha_3 = -\frac{(h_{n-1}+h_n)h_{n-1}}{(2h_{n-1}+h_n)}.$$

If the test problem of Eqn (6.20) is solved using Gear-2, the results given in Fig. 6.5 are obtained.

Applying the Gear-2 integration method to Eqn. (6.19), the expression used in the SPLAX program is obtained:

$$\left[\frac{1}{\alpha_3}\frac{\partial q(v_{n+1}^k)}{\partial v}-\frac{\partial f(v_{n+1}^k)}{\partial v}\right](v_{n+1}^{k+1}-v_{n+1}^k) \tag{6.21}$$

$$= f(v_{n+1}^k)-\frac{1}{\alpha_3}q(v_{n+1}^k)-\frac{\alpha_1}{\alpha_3}q(v_n^k)-\frac{\alpha_2}{\alpha_3}q(v_{n-1}^k)$$

$$-\left[\frac{\alpha_1}{\alpha_3}\frac{\partial q(v_n^k)}{\partial v}(v_n^{k+1}-v_n^k)\right]-\left[\frac{\alpha_2}{\alpha_3}\frac{\partial q(v_{n-1}^k)}{\partial v}(v_{n-1}^{k+1}-v_{n-1}^k)\right]$$

One drawback of the Gear-2 method is that it has twice the LTE of the trapezoidal method and this may result in slightly more time-steps than the trapezoidal method over a given interval, for the same accuracy. However, the disadvantage of trapezoidal point-to-point ringing, which is certain to occur due to the nature of the time-step control scheme used in WRN, outweighs its advantage and for this reason the Gear-2 method is preferred.

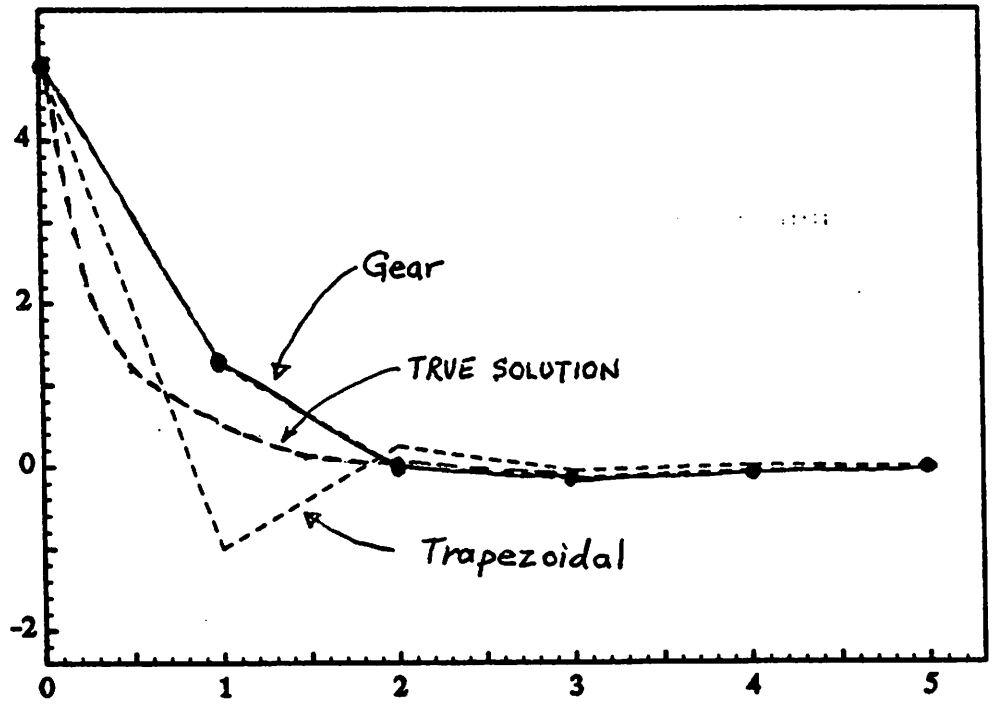


Figure 6.5 Gear-2 Integration Method
"$x_0 = 5.0 \quad |h\lambda| = 3$"

6.4.3. Waveform Limiting Techniques

The WRN approach may encounter some problems in handling circuits containing highly nonlinear elements such as diodes in the forward-bias region of operation. This problem is illustrated in Fig. 6.6(a) for a simple resistor-diode circuit with a grounded capacitor. Fig. 6.6(b) shows the waveform iterations obtained using WRN to solve the circuit. If the initial guess is $v(t)=0$ for all t, the first computed waveform is far from the correct solution. Due to the exponential nonlinearity of the diode, the waveforms converge very slowly to the final solution. In fact, over 50 iterations are necessary to converge in this example.

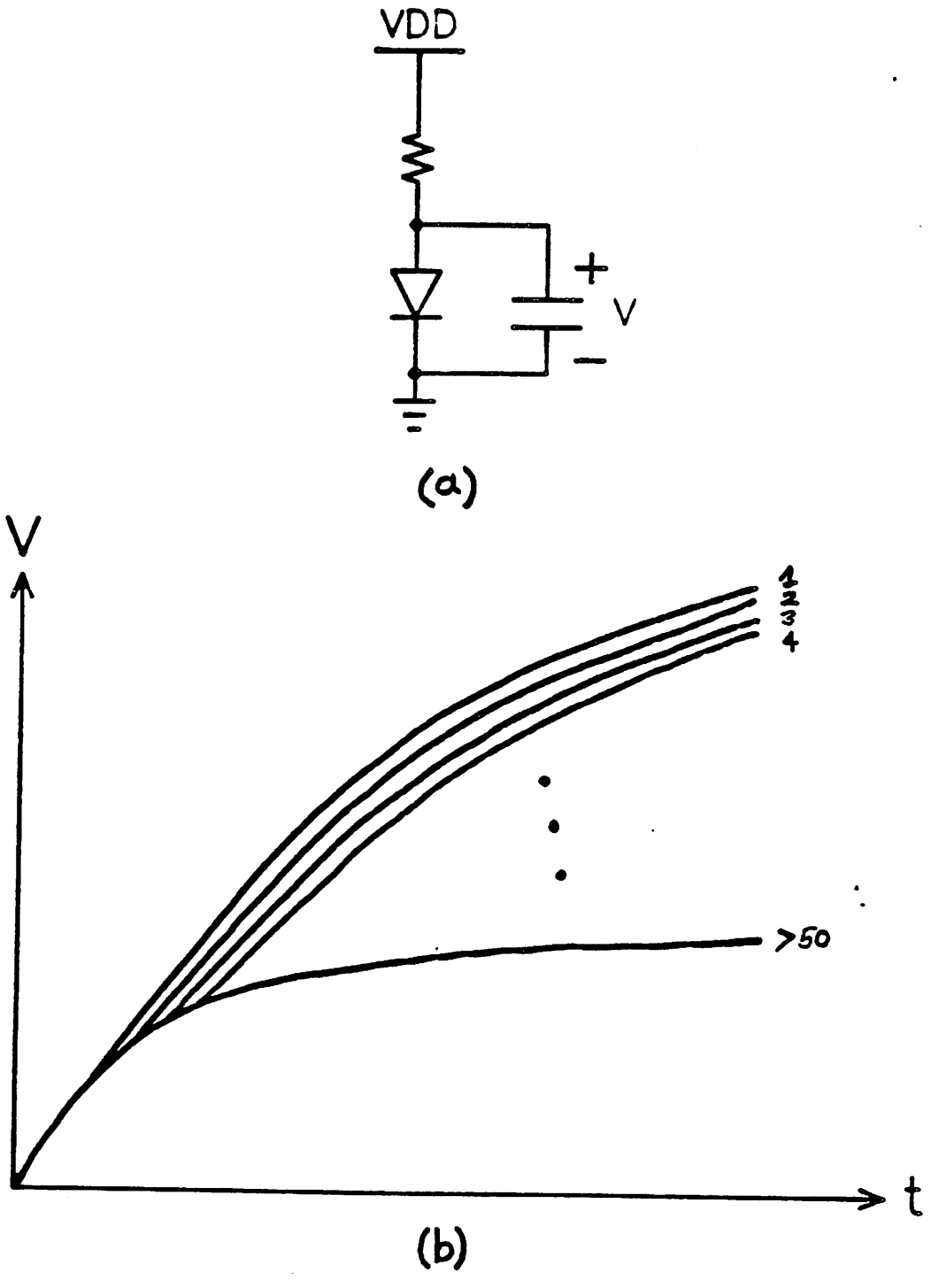


Figure 6.6 Diode Example

There are a variety of limiting techniques which can be used to improve convergence in this situation. The simplest approach is to use waveform limiting to ensure that the waveform does not take unrealistic excursions as time increases. For example, in a circuit simulation application, the maximum change allowed on each waveform iteration could be set to the value of the power supply. Another possibility is to perform limiting on the nonlinear elements in the circuit. Since the diodes associated with the source and drain of MOS transistors usually operate in the reverse bias region, a simple approximation can be used in the forward bias region to compute its conductance value. In particular, a line-through-origin model can be used for the diode whenever it switches into the forward bias region during the iterative process. Of course, only the Jacobian matrix is updated using this line-through-origin conductance value. The RHS term is always evaluated using the correct value of the current for the diode in case the true solution lies in the forward bias region of operation.

Another approach is to use smaller windows to control the waveform excursions. This would also improve the speed of convergence of the WRN method. However, some of the advantages of WRN with respect to multirate exploitation would be lost. Proper limiting would allow larger windows to be used but improvement in run time is likely to depend on the particular circuit being simulated.

6.4.4. Simulation Results

A preliminary version of WRN has been implemented in SPLAX. It uses the recursive divide-by-two time-step control, Gear-2 integration method and the waveform limiting techniques described in the previous sections. In Table 6.1 below, global-variable time-step ITA, standard WR and WRN are compared using three example circuits. The first example is a three-stage ring oscillator circuit. A substantial amount of floating capacitance makes the three inverters moderately coupled, but

because the three nodes are oscillating at the same frequency, the circuit does not exhibit the multirate property. For this example, the ITA algorithm is more efficient than WR since iterations are much cheaper in ITA. Furthermore, the ring oscillator represents the worst-case situation for WR [Lel81] since there is a feedback path connecting the output of the last inverter to the input of the first gate. Therefore, the number of waveform iterations necessary for this example is proportional to the number of cycles in the output waveform in each window. If the windows are too large, the waveforms will converge very slowly.

The second example is a critical path from a microprocessor and the third example is the logic for a successive approximation register. These two circuits are moderately coupled and exhibit substantial multirate behavior as shown in Chapter 3. As expected, for these two circuits WR is more efficient than global-variable time-step ITA. Because WRN exploits multirate behavior, and has a low cost per iteration, the WRN algorithm is more efficient than either WR or ITA in the cases shown below. However, for unidirectional circuits the WR algorithm would be more efficient and for highly nonlinear circuits both ITA and WR are expected to perform better than WRN.

Circuit	Mosfets	Nodes	SPLICE3.1	RELAX2.3	WRN
RINGOSC	7	3	9.8	27	7.2
DECPLA	116	66	194	160	137
SCDAC	344	151	1025	1010	618

Table 6.1 : SPLICE3.1 vs RELAX2.3 vs WRN

CPU-time in sec. on Vax11/785 under UNIX

6.5. CONCLUSIONS

The Waveform Relaxation-Newton method is an extremely efficient way to exploit the multirate property of circuits. The reason for this can be understood by examining the relationship between the three relaxation methods implemented in the SPLICE3.1,

RELAX2.3 and SPLAX programs, respectively. The outer loop for all methods is the Gauss-Seidel relaxation iteration. The ITA method uses a single Newton iteration to approximate the solution of each nonlinear equation in the inner loop. Similarly, WRN uses a single Waveform-Newton iteration to approximate the solution of each differential equation in the inner loop. For standard WR, an accurate (but potentially expensive!) method is used to solve each differential equation in the inner loop. However, larger windows may be used in WR and this allows multirate behavior to be exploited more effectively. The window size for WR as implemented in RELAX2.3 is some fraction of the total simulation period, i.e., T_{stop}/N. In SPLAX, the window size is some multiple of the user requested plot increment, i.e., $k*T_{plot}$, and this is usually smaller than the window size used in WR. The advantage of smaller windows is that the waveforms converge more rapidly. The global-variable time-step ITA method uses even smaller window sizes, equal to the global step sizes, but multirate behavior is not exploited in this case.

To summarize, WRN exploits multirate behavior using the WR approach and each iteration is relatively inexpensive as in ITA. The windows are "medium" size which results in faster convergence compared to WR at the expense of some multirate exploitation. In addition, the time-step control is such that very little work is done per iteration when the waveforms are far from the correct solution, but as the waveforms approach convergence the computational effort increases proportionately in order to produce accurate solution. Based on this comparison, it is not surprising that WRN outperforms the other two relaxation methods.

CHAPTER 7
PARALLEL ASPECTS OF ITERATED TIMING ANALYSIS AND WAVEFORM-NEWTON

7.1. INTRODUCTION

To this point, the algorithms described in this dissertation have been intended for computers which use a single central processor. These algorithms are referred to as *sequential* algorithms. The speed improvement obtained in the sequential Iterated Timing Analysis (ITA) and Waveform-Newton (WN) algorithms, with respect to the direct methods, were due primarily to exploitation of the latency and multirate properties of the circuit under analysis, and improvements in the time-step control. Further speed improvement can be obtained using parallel processors to exploit the natural decomposition of relaxation algorithms. A number of relaxation-based techniques have already been implemented on multiprocessors. In particular, the ITA algorithm used in SPLICE1.7 has been implemented in the MSPLICE program [Deu84,Jac86] on the BBN Butterfly [Ret79]. The Waveform Relaxation method has been implemented in the PRELAX program [Whi85c] on the Sequent multiprocessor [Seq84] and a nonlinear relaxation scheme has been implemented [Web87] on the highly-concurrent Connection Machine [Hil81]. Recently, the CONCISE program [Mat86], which also uses Waveform Relaxation, has been described for use on the Cosmic Cube [Sei85].

In this chapter, parallel aspects of the ITA and Waveform-Newton methods are explored further. In Section 7.2, the basic concepts of parallel computation are briefly described. In Section 7.3, synchronous and asynchronous relaxation methods are described. In Section 7.4, a number of issues concerning the implementation of ITA on a multiprocessor are described including a description of MSPLICE and a new program

called PSPLICE. In Section 7.5, a number of ways of parallelizing the WRN algorithm are proposed. To conclude the chapter, a generalized space-time scheduling model is described in Section 7.6.

7.2. BASIC CONCEPTS OF PARALLEL COMPUTATION

7.2.1. Classification of Computers

Digital computers are usually classified into four groups based on the multiplicity of instruction and data streams [Fly72]. These classifications are given by the acronyms SIMD, MIMD, MISD and SISD. Uniprocessors are usually considered to be SISD (single-instruction, single-data) machines whereas multiprocessors are usually classified as either MIMD (multiple-instruction, multiple-data) or SIMD (single-instruction, multiple-data) machines. SIMD machines are characterized by multiple processing units supervised by the same control unit. All processing units execute the same instruction but operate on different data from distinct data streams. Of primary interest here is the MIMD machine which is typically a collection of processors, each with their own local memory and some method of communicating with one another. Each processor has distinct instruction and data streams and the machine operates as a collection of SISD machines. An MIMD machine is considered to be *tightly-coupled* if the interaction between processors is relatively high. Otherwise, it is considered to be a *loosely-coupled* machine. Some architectures are hybrids which promote the use of small clusters of tightly-coupled processors with relatively loose coupling between the clusters [Kuc80].

7.2.2. Communication Between Processors

Communication between processors is an important aspect of a multiprocessor system. The underlying hardware support for communication may be organized in a

number of ways. For example, it may be a shared-bus, a crossbar switch, a multiport memory, or one of the many multistage interconnection networks (e.g. delta network, omega network, hypercube). For details on each type of network, see [Ens77]. Typically, communication is performed using either message-passing or shared data. Message-passing is usually found on relatively loosely-coupled machines whereas the shared-memory approach is employed on tightly-coupled machines. However, the user-level view of communication may be quite different from the hardware-level implementation of communication. For the user, a shared memory model of communication simplifies program development. Therefore, some multiprocessor systems give the user the illusion of shared-memory even though the underlying implementation may, in fact, be based on message-passing. Of course, the purpose of the user-level view is to convey information about the relative cost of message-passing vs. sharing data. Whether a particular user-level view is appropriate or not depends on how expensive a *remote memory reference* is relative to a *local memory reference*. If the local memory reference time is very small relative to the remote memory reference time, the message-passing model for communication is more appropriate. If not, the user-level shared data model is more appropriate. To illustrate the two communication approaches further, consider the examples of the BBN Butterfly and the Sequent multiprocessors.

The Sequent is a tightly-coupled multiprocessor which employs a shared-bus and shared-memory architecture. As shown in Fig. 7.1, the processors, memory and I/O devices are all connected to a single, common bus. Processors obtain instructions and data from the large pool of memory using the common bus. Therefore, the bandwidth of the bus must be high enough to handle simultaneous requests from all processors and I/O devices. One problem with this type of architecture is that it does not easily accommodate growth. Adding extra processors to the system will eventually degrade

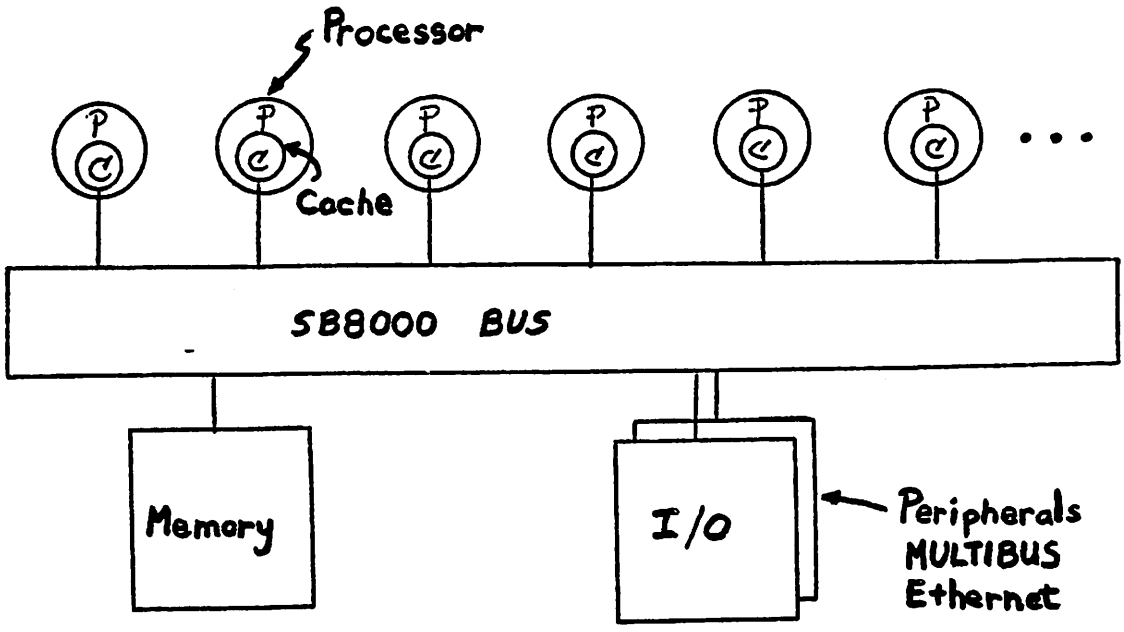


Figure 7.1 : Architecture of the Sequent Machine

the overall performance. In order to improve the performance, a large cache is provided on each processor to reduce bus traffic. The cache contains the most recently used blocks of memory and exploits the locality of reference property of most programs. Since there is a separate cache for each processor in the system, the problem of maintaining consistency between the various caches and main memory exists. On the Sequent, a *Write-Through with Invalidation* scheme [Fie84] is used, as described in the next section. In view of the memory hierarchy on the Sequent, a local memory reference can be considered as one which is made to the cache whereas a remote memory reference is one made to main memory. These are only conceptual definitions as there is little in the way of direct control of the number of each type of reference.

The Sequent allows an executing user process, called a *parent* process, to create subprocesses, called *child* processes, which execute on any of the available processors. Here, it is assumed that there is a one-to-one correspondence between the number of processes and the number of processors, although this rule is not strictly enforced on the Sequent[1]. Therefore, the terms "process" and "processor" are used interchangeably here. The parent process uses the UNIX *fork* command to create child processes. During the creation procedure, each child process obtains a copy of the parent's entire virtual address space, and this is viewed logically as local memory for each process. Each process also has access to a pool of shared memory. Interprocessor communication on the Sequent is performed via shared memory. A set of shared variables may be defined by the parent process for purposes of synchronization and exchange of information. However, special care must be taken when a number of interacting processes are updating global data, and this aspect is described in a section to follow. For the Sequent, the communication rate is of the order of the memory bandwidth and all memory references require roughly the same length of time.

The BBN Butterfly multiprocessor [Ret79] has a distributed memory system. The architecture of the Butterfly is shown in Fig. 7.2. Each processor has its own local (physical) memory and the processors are connected using a high-speed Omega network [Law75]. An example of a 3-stage Omega network is shown in Fig. 7.3. It is configured as a *shuffle-exchange* network [Sto71]. The shuffle operation, at each stage, divides the available connections in half and rearranges them as if they were shuffled perfectly (like taking a deck of cards, cutting them in half and shuffling them evenly). The output of each shuffle stage feeds a set of exchange boxes. Each exchange box has two inputs and two outputs and is capable of either passing information straight through or

[1] The Sequent is a multiuser machine which performs automatic dynamic load balancing and allows the number of processes to exceed the number of available processors.

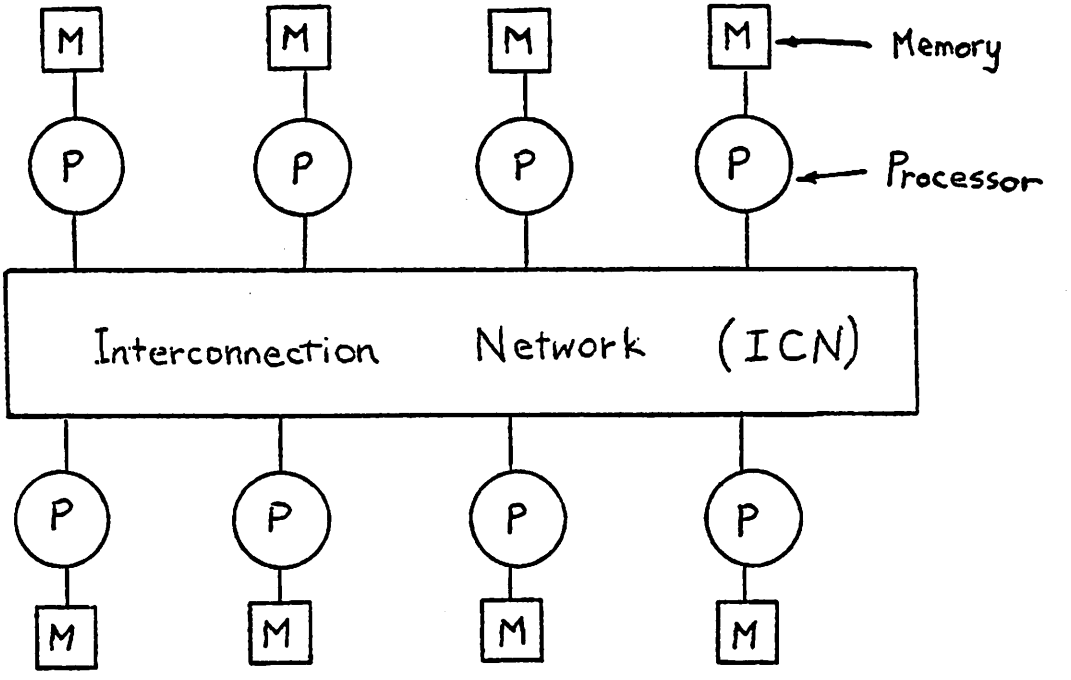


Figure 7.2 : Architecture of the BBN Butterfly

exchanging information from a given input to the alternate output. Eight processors, numbered 0 through 7, can be interconnected using this 3-stage network. The processors would be connected in two places, as indicated in the figure, at both ends of the network. A given processor communicates with another processor via the shuffle-exchange network by transmitting the address of the destination along with the data. At each stage of the network, the message is either passed directly through the exchange boxes or switched to the alternate path, depending on the value of the active bit of the address. Since there are $\log N$ stages in the network, where N is the number of processors in the system, the communication time is proportional to $\log N$.

For the Butterfly, a local memory reference is one that is made in the processor's own local memory and is relatively fast. A remote memory reference is one that refer-

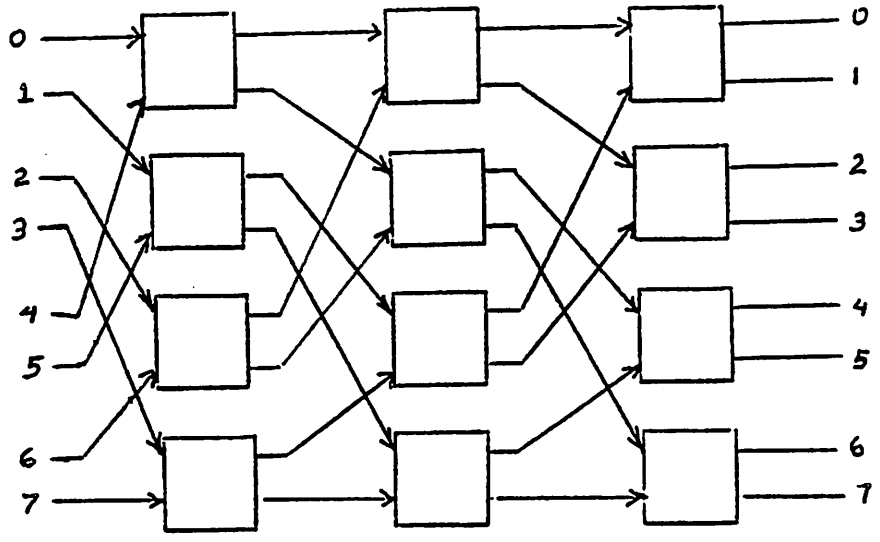


Figure 7.3 : 3-Stage OMEGA network

ences a memory location associated with another processor over the network and is relatively slow — approximately five times slower than a local memory reference. If the computation time for a task is short relative to the communication time, special care must be taken to minimize the number of remote memory references used in a program. Low-level communication between processors on the Butterfly is done using message-passing. For example, if processor p_i requires data from processor p_j, it sends a message to p_j via the interconnection network. Processor p_j then sends the appropriate information back to p_i in a similar manner. The operating system on the Butterfly allows the user to define the protocol for message-passing by providing a set of communication primitives.

A shared memory model of communication is also supported on the Butterfly via the Uniform System [BBN85]. From a programming standpoint, there is little difference between this approach and the approach used on the Sequent. However, the underlying implementation is quite different. Whenever a shared variable is declared on the Butterfly, it is defined uniquely in one memory location on one of the processors in the system. If another processor requires access to this shared variable, it must perform a remote memory reference to the appropriate processor. Therefore, accessing shared variables is more expensive than accessing non-shared variables. In some sense, local memory can be viewed as a cache for non-shared variables whereas the shared variables are viewed as being in "main memory" and requires a longer access time. The performance of any system may be degraded if many processors attempt to access a given shared variable. To reduce problems of contention on the Butterfly, shared data is scattered uniformly throughout the machine and redundant paths are provided in the interconnection network to minimize traffic congestion.

7.2.3. Mutual Exclusion

The use of shared variables or shared data structures between processors requires some mechanism to establish mutual exclusion. This ensures that a common resource is held by only one processor at a time. Mutual exclusion is usually implemented using an atomic (or indivisible) operation, such as the *test-and-set* synchronization primitive. The mutual exclusion policy is enforced by requiring that each process successfully execute the atomic operation before gaining access to a shared variable. A shared variable is usually updated in a *critical* section of the program. A critical section is defined as any segment of a program which may only be executed by one processor at a time. A process entering such a section, S_i, must first *lock* the section before entering it using the test-and-set instruction, and then *unlock* the section upon exiting it as follows:

```
.
.
LOCK(lock_variable )
execute critical section S_i ;
UNLOCK(lock_variable )
.
.
.
```

The *lock_variable* argument is set when the *LOCK()* routine is executed successfully and reset when the *UNLOCK()* routine is executed. Other processes trying to enter the critical section are placed in a queue and are required to wait until the unlock operation is performed.

If a critical section is long and many processes attempt to enter the section simultaneously, the system could be highly underutilized since most of the processors would be busy accessing and testing the lock variable until the critical section becomes free. This is referred to as *busy-waiting*. The time that a processor spends busy-waiting is essentially wasted and results in a loss of efficiency. Another source of performance degradation due to locks is the number of memory references generated while repeatedly accessing the lock variable. If a large number of processors are busy-waiting on the same lock variable, the communication traffic generated may be enormous. This would have a major impact on the performance of a shared-bus system such as the Sequent. To avoid this problem, the Sequent uses a Snooping Cache strategy [Kat85] by enforcing a *Write Through with Invalidation* protocol [Fie84]. That is, if a processor wishes to write to an address in its cache, the data is also written into main memory. This is the Write Through aspect of the cache consistency scheme. Other caches continuously monitor the addresses of all write operations to main memory. If a particular write address also exists in a given cache, the associated data in that cache is invalidated. As a result, the bus traffic is a function of the number of write operations to main memory and not the total number of memory references. Therefore, busy-waiting does not generate excess bus traffic once the lock variable is in a processor's

cache memory.

7.3. SYNCHRONOUS AND ASYNCHRONOUS RELAXATION

In the implementation of iterative algorithms on multiprocessors, one must consider whether to use a synchronous or asynchronous algorithm. If synchronization is performed after each relaxation iteration, the algorithm is categorized as a *synchronous relaxation* method. Here, synchronization implies that the processes must wait until they all have completed their tasks in the current iteration before proceeding to the next iteration. In general, if a parallel algorithm consists of a number of "cooperating" processes and *any* process waits for another process to finish its task before proceeding with the next task, the algorithm is referred to as a *synchronized algorithm* [Kun76]. Synchronization points are normally used to update global information, to exchange information between processes, and to assign the next set of parallel tasks. The penalty for synchronization depends on the number of processors in the system [Kun76]. If many processors are waiting at the end of each iteration for others to finish the efficiency can be degraded significantly. Hence, the synchronous approach is not well-suited to parallel computation on large machines due to potential losses in efficiency and surges in communication traffic when exchanging information.

An alternative to the synchronous approach is to use one of the *asynchronous relaxation* schemes. One such scheme, called "chaotic" relaxation, was suggested by Chazan and Miranker [ChMi71]. In this scheme, the new values of a given component are computed using whatever values are available for the external variables. That is, the computation is not required to follow any particular order nor is a given component required to use a particular set of previous iterates in its evaluation. In fact, the external values could be recently computed values or values from earlier iterations. The advantages of this approach are that it removes synchronization from the iterative

process and that it distributes the communication load more evenly over time.

A general model for asynchronous computation was introduced to provide a framework for the description and analysis of asynchronous algorithms [ChMi71]. The model given here uses Baudet's notation [Bau78] (which in turn is based on Chazan and Miranker [ChMi71]).

Definition 7.1: (Asynchronous Computation Model - ACM)

Given the fixed-point problem $x = F(x)$ where $F:\mathbb{R}^n \rightarrow \mathbb{R}^n$ with components $f_i(x), i=1,....n$ and $x^k \in \mathbb{R}^n$, and a starting vector x^0, an asynchronous iteration is defined by the update equation:

$$x_i^k = \begin{cases} f_i(x_1^{k-s_1}, \cdots, x_n^{k-s_n}) & i \in J_k \\ x_i^{k-1} & i \notin J_k \end{cases} \tag{7.1}$$

The update equation is subject to the following conditions:

(a) $s_i \geq 1$

(b) $k - s_i \rightarrow \infty$ as $k \rightarrow \infty$

(c) i occurs an infinite number of times in J_k. $k = 1,2,....$

■

This model is simply an update equation for a variable x_i at the k th iteration. As seen in Eqn (7.1), x_i is computed using the function f_i, if it is in the update set, J_k, or simply updated using the value which existed previously, if it is not in the update set. The update set for the k th iteration is given by J_k and is comprised of the set of components to be computed at some point in real time T_k during the execution of the parallel solution. As will be seen in an example to follow, the iteration index k is derived from the real time point T_k at which a particular computation is started. These points in real time should not be confused with simulation time points which would be specified as t_k for time point k. The values used to compute f_i are specified in the form $x_m^{k-s_m}$. The variables, s_m, represent delays in the iteration values of the other components. The conditions associated with the ACM are intuitively obvious. Condition (a) requires that only values from some previous iteration (i.e., values computed at previous points in real time) be used to compute each x_i. Condition (b) requires that

old values cannot be used indefinitely and that newly computed values must be used at some point during the computation. Condition (c) requires that no component be abandoned forever.

A variety of relaxation schemes, such as Gauss-Jacobi, Gauss-Seidel, and chaotic relaxation, can be described in the ACM by selecting the appropriate values for J_k and the s_m's [ChMi71, Bau78]. For example, the Gauss-Jacobi method is obtained by setting $J_k = \{1,2,3,....,n\}$ for $k=1,2,....$ and $s_m=1$ for all $m=1,....,n$. An example using the asynchronous computation model is given in Fig. 7.4. There are three variables in the system, x_1, x_2 and x_3, and three processors P_1, P_2 and P_3. Each variable, x_i, is assigned to a particular processor, P_i, as shown along the y-axis. The x-axis is the execution time for the problem and each point at which a computation ends corresponds to an iteration. At time point 0, all three processors begin to compute new values of their assigned variables using the initial guess x^0. At time T_1, $J_1=\{1\}$ and the update equations are given by:

$$x_1^1 = f_1(x_1^0, x_2^0, x_3^0)$$

$$x_2^1 = x_2^0$$

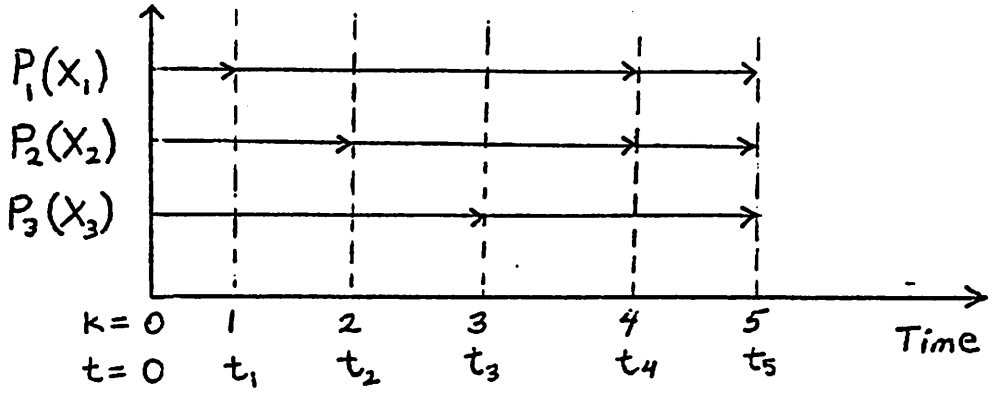
$$x_3^1 = x_3^0$$

Other update sets and update equations are given in the figure. The update equations use the most recently available values of the components, although any choice which satisfies the conditions of the ACM can be used in the computation.

While the asynchronous approach seems to be an attractive alternative to the synchronous approach, it is important to ask whether or not the iterative method is still guaranteed to converge. Using the problem:

$$x = F(x) \tag{7.2}$$

Baudet proved the following theorem [Bau78]:



$$J_0 = \{1, 2, 3\}$$

$$J_1 = \{1\}\ x_1^1 = f_1(x_1^0, x_2^0, x_3^0),\ x_2^1 = x_2^0,\ x_3^1 = x_3^0$$

$$J_2 = \{2\}\ x_1^2 = x_1^1,\ x_2^2 = f_2(x_1^0, x_2^0, x_3^0),\ x_3^2 = x_3^1$$

$$J_3 = \{3\}\ x_1^3 = x_1^2,\ x_2^3 = x_2^2,\ x_3^3 = f_3(x_1^0, x_2^0, x_3^0)$$

$$J_4 = \{1, 2\}\ x_1^4 = f_1(x_1^1, x_2^0, x_3^0),\ x_2^4 = f_2(x_1^1, x_2^1, x_3^0),\ x_3^4 = x_3^3$$

$$J_5 = \{1, 2, 3\}\ x_1^5 = f_1(x_1^4, x_2^4, x_3^3),\ x_2^4 = f_2(x_1^4, x_2^4, x_3^3),\ x_3^4 = f_3(x_1^1, x_2^2, x_3^3)$$

Figure 7.4 : Example of the Asynchronous Computation Model

Theorem 7.1: (Convergence of Asynchronous Iterations)

If F is a contracting operator in a closed region D of $\mathbb{R}^n$ and if $F(D) \subset D$, then any asynchronous iteration corresponding to F with an initial guess $x^0 \in D$ which satisfies the conditions of the ACM converges to the unique fixed point of F in D.

■

Therefore, the asynchronous approach is a viable alternative to the synchronous approach.

7.4. PARALLEL ITA ALGORITHMS

The sequential version of the ITA algorithm was described in Chapter 4. In this section, two parallel implementations of this algorithm are described. One straightforward way to parallelize ITA is to perform the tasks in each iteration in parallel, as follows:

Algorithm 7.1 : (A Simple Parallel ITA Algorithm)

```
t_n ← 0;
h_next ← h_min;
while (t ≤ T_stop) {
    t_n ← t_n + h_next ;
    k ← 0;
    repeat {
        forall ( i ∈ 1, ··· ,n ) { /* perform in parallel */
            solve J_Fi(v^{k,i})(v_i^{k+1} − v_i^k) = − F_i(v^{k,i}) for v_i^{k+1};
        }
        k ← k +1;
    } until ( ||v_i^{k+1} − v_i^k|| <ε_1 , ||F_i|| <ε_2, i =1...n )
    h_next ← PickStep();
}
```

■

where $F_i(v)$ and $J_{F_i}(v)$ have been specified previously in Eqns. (4.3) and (4.5). The use of the **forall** construct in Algorithm 7.1 implies that all n equations in the system can be solved in parallel. The basic steps in the algorithm are the same as described in

Chapter 4. Initially a time step is selected and the active subcircuits are scheduled at the first time point. The subcircuits are processed in parallel producing new solutions for iteration 1. The solutions are exchanged and the next iteration, iteration 2, is carried out in the same fashion. This continues until the iterations converge to the solution. Once convergence is obtained at the time point, a new step size is selected and the same steps are carried out again at the next time point. The details of the time step control and latency exploitation have been omitted in the algorithm to simplify the description.

There are two forms of synchronization in Algorithm 7.1. One type occurs after each iteration and the other after obtaining the solution at each time point. Synchronization after each iteration guarantees that the s_j's are bounded by 1 for all j in the asynchronous model given in Eqn. (7.1), i.e., the "chaos" is bounded by one iteration. However, synchronization should be avoided, if possible, for reasons given earlier. Therefore, both MSPLICE and PSPLICE use weakly chaotic relaxation, implemented using event-driven techniques, as described in the sections to follow.

7.4.1. MSPLICE - A Multiprocessor Implementation of SPLICE1.7

MSPLICE [Deu84] is a parallel implementation of the ITA algorithm used in SPLICE1.7 on the Butterfly multiprocessor. MSPLICE uses *data partitioning* where each processor performs identical functions on different parts of the circuit rather than *functional partitioning* where different functions would be assigned to different processors. A task in MSPLICE is defined as the evaluation of a single variable (in this case, a node voltage) for a single iteration. Therefore, the granularity of each task is approximately the same and this is desirable from the standpoint of load balancing. Since there are usually many more nodes in the circuit than there are processors in the system, more than one node may be assigned to each processor. A distributed scheduling algorithm is

used for task assignment whereby each processor is responsible for scheduling not only its own tasks but also certain tasks for other processors.

MSPLICE uses a weakly chaotic relaxation scheme to avoid synchronization during the iterative process. A global convergence counter, *GlobalRemainingNets*, is used to coordinate the processors at each time point. This variable is incremented whenever a node is scheduled for an iteration and decremented when the node is processed. When *GlobalRemainingNets* reaches zero, it indicates that all nodes have converged at the present time point. Therefore, the processors may proceed to the next time point. In MSPLICE, nodes can be assigned to processors using either static or dynamic allocation and a number of tradeoffs exist in each approach [Deu85, Jac86]. For example, in a static allocation scheme, the architecture of the Butterfly suggests that adjacent nodes in a circuit be assigned to the same processor to keep the number of remote memory references to a minimum. However, latency exploitation usually reduces the number of tasks available during the iterative process. Since adjacent nodes in the circuit tend to be latent at the same time, some processors may run out of tasks while others still have many tasks to process. This creates an imbalance in the processor loads and suggests that adjacent circuit nodes should be placed on different processors to reduce the likelihood of all the nodes on one processor being latent.

A dynamic assignment strategy is preferable to a static assignment so that load balancing can be performed more effectively during execution. However, a significant overhead may exist in moving circuit data from one processor to another, unless a copy of the entire circuit exists on each processor, which is memory inefficient for large circuits. Another issue in dynamic assignment is that of performing the dynamic load balancing. In a more recent version of MSPLICE [Jac86a], each processor schedules new events on the processor with the fewest events in its queue. This involves searching a

global data structure to determine the processor that has the shortest task queue. As the number of processors increases, contention for the global data structure could degrade the performance of the program. In fact, since N entries in the global data structure are searched by a possible N processors, the overall contention grows as N^2 in this implementation. However, since a task requires a significant amount of time to perform, the queue contention does not pose a major problem with up to 100 processors [Jac86]. In the current implementation, the dynamic scheduling algorithm shows the best performance [Jac86], although the program allows the user to select the mode of task assignment for experimentation purposes.

7.4.2. PSPLICE - A Parallel Implementation of SPLICE3.1

In this section, a number of different approaches are described for parallelizing ITA on a shared-bus, shared-memory architecture where a limited number of processors are available. The Sequent machine was used as the test-bed multiprocessor. The basic algorithm used was the global-variable step ITA with partitioning as implemented in SPLICE3.1. The issues to be examined are: central scheduler vs. central queue, the granularity of the computation, Gauss-Seidel/Gauss-Jacobi alternatives, and synchronization at iteration and time point boundaries.

a. Tasks and Task Scheduling

In the implementation of a parallel algorithm, it is necessary to first create the processes which will perform tasks concurrently. On the Sequent, this is implemented using the UNIX *fork()* system call. Normally, a corresponding *join()* operation would be used upon completion of the tasks to remove the processes which are no longer useful. Another set of forks would then be issued to perform a different set of tasks in parallel. Unfortunately, a UNIX *fork()* operation is rather expensive [Seq84] and if this

system call is used every time an opportunity for concurrency arises, the overhead could degrade the performance of the machine. One way to avoid this problem is to perform the forking operation in one loop in the program (to create a separate process for each available processor). Then, whenever a set of tasks become available for concurrent execution, they are assigned to the existing processes. Each process performs the assigned task and then simply waits for the next task rather than performing a *join()* operation. A *task*, in this context, has two components: a function (implemented as a function pointer in C), and a piece of data to be operated on by the function (implemented as simple pointer in C). Using this approach, each processor may perform a number of different functions during the execution of the program without incurring the additional penalty of the forking operation.

The actual assignment of tasks to processors is implemented using a scheduling operation. Two approaches were implemented to perform dynamic task scheduling:

(1) a central scheduler,

and (2) a central queue

The central scheduler approach requires one process to manage the scheduling of tasks and perform certain bookkeeping functions. It can be implemented using *mailboxes*, which are simply message-buffers for interprocess communication:

Algorithm 7.2 : (Central Task Scheduling Approach)

```
parent() {
    set-up p mailboxes;
    fork p processes which execute child() routine;
    AllDone ← TRUE;
    task ← CreateNextTask();
    repeat {
        AllDone ← FALSE;
        while ( task ≠ NULL ) {
            foreach ( i ∈ 1, ··· ,p ) {
                if ( mailbox[i] empty ) {
                    mailbox[i] ← task;
                    task ← NULL;
                    break;
                }
```

```
            }
        }
        task ← CreateNextTask();
    } until ( task = NULL );
    AllDone ← TRUE;
}

child(i) {  /* each child is assigned a process number i */
    repeat {
        while ( mailbox[i] empty && AllDone = FALSE ) { wait here };
        if ( AllDone = FALSE ) {
            task ← mailbox[i];
            Execute(task);
            clear mailbox[i];
        }
    } until ( AllDone );
    exit;
}
```

In this implementation, the parent process is responsible for creating the tasks in some manner and then assigning them to idle processors which are identified by scanning the mailboxes. The child processes check their respective mailboxes for a task, and then execute the task when it arrives. This approach is depicted in Fig. 7.5.

The problem with this approach is that for a small number of processors, the system may be under-utilized since the parent is dedicated to the management of tasks. For a large number of processors, the parent may become a bottleneck if it cannot generate and assign tasks fast enough to keep all the processors busy. The advantage of this approach is that fewer synchronization operations, involving locking and unlocking operations, are performed.

An alternative to this approach is to use a central queue. In this approach, the parent process sets up a queue of initial tasks and then executes the same piece of code as the other processes -- effectively becoming a child process itself. The tasks are then removed one-by-one and executed by different processors. At the same time, new tasks are added to the queue by the processors upon completion of old tasks. This approach is

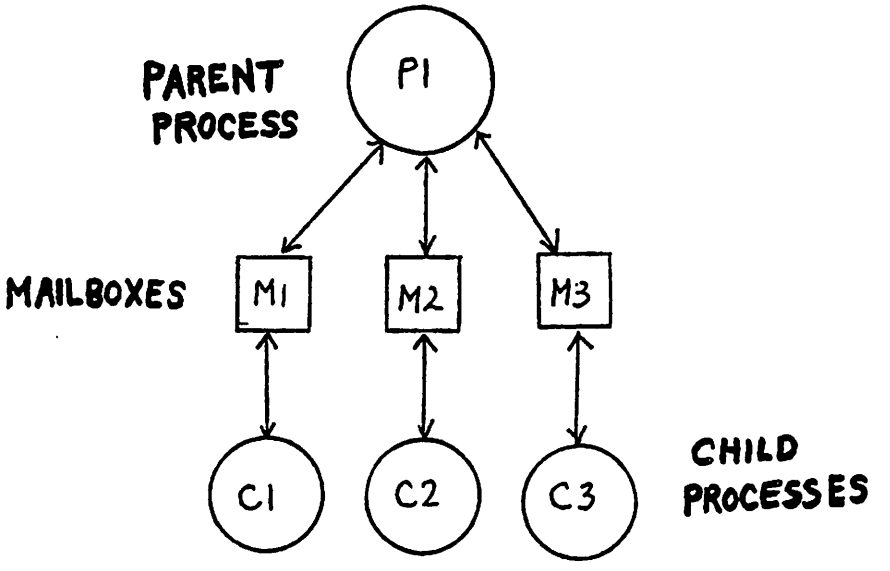


Figure 7.5 : A Central Scheduler Approach

depicted in Fig. 7.6 and the algorithmic details are given below:

Algorithm 7.3 : (Central Queue Scheduling Approach)

```
setup() {
    set-up queue Q;
    numberOfBusyProcessors ← 0;
    fork p processes which execute child() routine;
    child();
}

child(i) {  /* each child is assigned a number i */
    repeat {
        lock(Q);
        task ← NextTaskFromQ();
        increment numberOfBusyProcessors;
        unlock(Q);
        if ( not NULL task ) {
            Execute( task );
            lock(Q);
            decrement numberOfBusyProcessors;
            AddNewTasksToQ();
            unlock(Q);
```

```
            }
            else {
                lock(Q);
                decrement numberOfBusyProcessors;
                unlock(Q);
            }
        } until ( Q empty && numberOfBusyProcessors = 0 );
        exit;
    }
```

■

Note that this approach is more uniform in nature than the central scheduler approach in the sense that the same routine is executed by all processes, rather than the parent-child routines executed in the previous case. This makes it somewhat easier to write the program (although neither approach is overly complicated) and promotes uniform growth if more processors are added to the system. The main advantage of the central

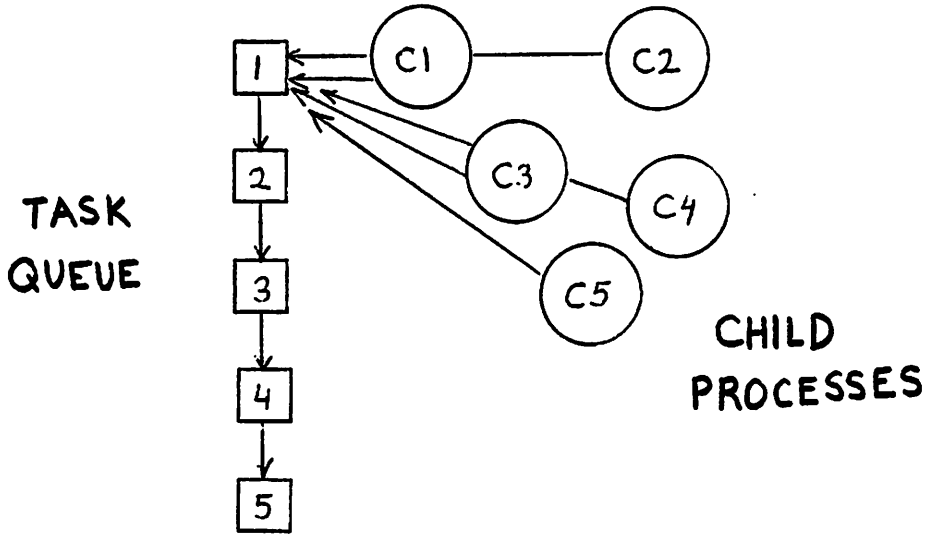


Figure 7.6 : A Central Queue Approach

queue is that all processors are actively involved in the solution to the problem and this improves the utilization. One drawback is that accessing the queue may become a problem when the number of processors is large, due to the lock/unlock operation. However, if the execution time of a task is much longer than the average waiting time for the queue, it is not a significant problem.

Both the central queue and central scheduler approaches were implemented in PSPLICE. It was found that the speed-up on two processors for the central queue method was close to 2.0 whereas the speed-up for the central scheduler case was close to 1.0 (since one processor was the parent whose job was to assign tasks while the other was the child performing the tasks). However, the speed-up was much closer on 8 processors, differing only by 10-15% in favor of the central queue approach. The central queue approach has been adopted in PSPLICE.

b. Granularity of the Computation

The granularity of the computation refers to the size of the tasks assigned to each processor in the parallel portions of a program. It seems natural to define a single Newton iteration for a subcircuit as the basic task in the parallel version of SPLICE3.1. However, this introduces a problem for load-balancing since some subcircuits may be very small, possibly containing a single node, while others may be very large containing many nodes. A processor which is assigned a small subcircuit would complete the task very quickly while another processor with a larger subcircuit may take much longer to complete its task. This nonuniformity of the task sizes may lead to underutilization, especially at synchronization points.

One way to resolve this problem is to create subcircuits which are roughly the same size. That is, after the usual partitioning step, a second pass could be performed to combine subcircuits together so that each group of subcircuits requires approximately

the same amount of time to evaluate. A task would then be defined as the evaluation of a cluster of subcircuits. The basic problem with this approach is that it reduces parallelism. In addition, some efficiency in latency exploitation is lost since more variables are solved together. However, this may be a still reasonable approach if the number of processors is small.

A better approach is to refine the granularity of the computation by breaking up the task of solving a subcircuit into a number of subtasks which can also be performed in parallel. For example, each subcircuit evaluation is composed of three subtasks: evaluation of the Jacobian and right-hand-side (RHS) vector entries, matrix loading, and matrix solution. Of these, the evaluation of the Jacobian and RHS vector, called model evaluation, usually dominates the run time for small subcircuits [New78]. Therefore, a good strategy is to allow both subcircuit evaluation and model evaluation to be performed in parallel. Of course, subcircuit processing would be reduced to setting up the tasks for model evaluation in a separate model queue. Other processes would remove model evaluation tasks from the model queue and compute and load the matrix entries. The last processor to complete a model evaluation associated with a particular subcircuit could also perform the matrix solution since it requires a relatively small amount of time. This approach is used in PSPLICE.

One problem with making each model evaluation a separate subtask is that some models are as simple as resistors or capacitors, requiring less than 10 floating-point operations, while others are complicated MOS transistors, requiring hundreds of floating-point operations. If a task is too small, the overhead of the queueing operation may be a significant portion of the time required to perform the task. A better approach is to cluster a number of circuit elements together such that each cluster requires approximately the same length of time to process, and that the processing time

is much greater than the overhead. The overhead due to queueing can be estimated by examining the associated operations. The queue is locked and unlocked once when the task is scheduled, and again when the task is removed for processing. Let the time required to perform these functions be T_{lock} and T_{unlock}, respectively, and let T_{CS} be the time that spent executing a *critical section* associated with the queue, such as enqueueing or dequeueing a task. If p processors are busy-waiting on the lock variable in the worst-case, then the execution time for the cluster, $T_{cluster}$, should satisfy the following condition:

$$T_{cluster} >> 2(T_{lock} + T_{CS} + T_{unlock})\, p$$

That is, the processing time should be much greater than the worst-case waiting time. For the Sequent, $T_{lock} + T_{unlock}$ is roughly $65\mu S$. If $p=8$ and T_{CS} is negligible, then $T_{cluster} >> 1000\mu S$. To create the clusters, the unit cost for each type of model evaluation must be known. This is listed in the table below for four different devices:

MODEL	TIME
MOS transistor	2000uS
Diode	600uS
Resistor	300uS
Capacitor	150uS

Table 7.1: Execution times for four devices

Using this table, the model evaluation subtasks can be generated so that they require more than $1000\mu S$. Note that if the clusters are too large, some degree of parallelism may be lost. Therefore, the actual size should be a parameter which can be tuned based on a few experiments. Currently, the task size is approximately $5000\mu S$ although slight variations in the size are permitted if necessary.

In the implementation of this approach in PSPLICE, the clusters are defined in a preprocessing operation. Two prioritized queues are used: one for subcircuit evaluation

and one for model evaluation. The model queue is given a higher priority than the subcircuit queue because it is preferable to generate new values for a subcircuit rather than starting the solution of another subcircuit. This increases the likelihood that the next subcircuit will get new values to compute its solution resulting in faster convergence. At the beginning of each time point, the model queue is empty and the subcircuit queue contains a list of tasks to be processed. When a subcircuit task is processed, a corresponding set of model evaluation tasks are scheduled in the model queue with the largest tasks first and the smallest tasks last. This gives the larger tasks a slight head start over the smaller tasks. Other processors obtain tasks from this model queue, since it has a higher priority than the subcircuit queue, until the queue is empty. The last processor to finish a model evaluation for a subcircuit performs the matrix solution. If it is anticipated that the matrix solution time will be large, it can also be decomposed into a number of smaller tasks and placed in a third queue which is of higher priority than either the model queue or subcircuit queue. However, this last feature has not been implemented in PSPLICE as yet.

c. Synchronization at Time Points

In PSPLICE, synchronization at iteration boundaries is avoided by using an event scheduler to generate tasks dynamically. That is, after processing a subcircuit, its fanout subcircuits are scheduled at the end of the queue if the subcircuit is "active". If the subcircuit has not converged, it is placed at the end of the queue. Therefore, each subcircuit in the queue has at least one input or internal variable which has changed in value. Processing continues in this manner, without synchronization, until the queue is empty, which indicates that the solution has been obtained at the current time point.

Synchronization at each time point is difficult to avoid since the next time-step is not known until all the variables have converged to the solution at the current time

point. In fact, neither MSPLICE nor PSPLICE allow the solution to proceed beyond a particular time point until the iterations have converged. However, the fixed time-step algorithm used in MSPLICE would allow nodes to go ahead to the next time point, if processors are available, since the next time-step is known. In the variable time-step case, the time step is selected by the fastest changing variable and this value is not known *a priori*. In principle, any step size which is smaller than the recommended step size may be used at each time point. This observation leads to a strategy to remove synchronization at time points in the variable step case. The idea is to use the step size chosen for the time point t_{n-1} at the next time point, t_n. For example, the time step, h_{n-1}, selected at t_{n-1} would be used at t_n rather than at t_{n-1}. A recursive relation defining the set of time points is given by:

$$t_{n+1} = t_n + h_{n-1}$$

By using this approach, the value of t_{n+1} is known while computing the solution at t_n. Since t_{n+1} is known in advance, any subcircuit that converges at t_n would be permitted to go ahead to the next time point, assuming that idle processors are available. The approach could be implemented using another set of queues for tasks at time point t_{n+1}. Whenever the task queues at t_n are empty, the idle processors could take items from the queues at t_{n+1} and process them even though the solution at t_n may not be completely finished. Therefore, some subcircuits would be scheduled at t_n while others would be scheduled at t_{n+1}. When processing a particular subcircuit at t_{n+1}, any values not available at t_{n+1} would have to be extrapolated. If a subcircuit which has been processed at t_{n+1} is subsequently re-scheduled at t_n, the solution for that subcircuit at t_{n+1} would simply be discarded.

In the implementation of this approach, two types of step rejections may occur: rejection of h_{n-1} and rejection of h_n. The rejection of h_{n-1} occurs if the iterations do not converge at t_n or if the solution is not accurate enough at t_n. The solution at both

t_n and t_{n+1} must be abandoned under these conditions and new values of t_n and t_{n+1} selected for subsequent processing. The other type of rejection occurs if h_{n-1} is accepted but h_n is considered to be too large. In this case, any further processing at t_{n+1} would be halted and the new value for h_n substituted for the one which was rejected.

d. Gauss-Seidel/Gauss-Jacobi Algorithms

One remaining issue to address is the order in which the subcircuit tasks should be processed. In a sequential implementation of a relaxation method, the components in the system are usually processed in a particular order and solved using the most recently computed values for all external variables, as specified by the Gauss-Seidel method. On a multiprocessor, the processing order and the external values used during the computation depend on the particular implementation. For example, if all the tasks are started simultaneously and the values for the external variables are always obtained from the previous iteration, the algorithm would be strictly Gauss-Jacobi. However, if the tasks are initiated whenever a free processor is available, then the method is chaotic relaxation. Both the Gauss-Jacobi method and chaotic relaxation offer more parallelism than Gauss-Seidel and result in higher processor utilization. However, the convergence speed is reduced and this may lead to increased runtimes. A number of experiments were performed using the PSPLICE program to compare these approaches.

The first step was to try a Gauss-Jacobi-like algorithm by processing as many subcircuits in parallel as possible. It will be referred to as the "First-Available-Task" approach. In this approach, the subcircuits are obtained from the head of the subcircuit queue and processed immediately using whatever values were available for external variables. While the method is **not** pure Gauss-Jacobi (since there may be updated information available in some cases) the approach is closer to Gauss-Jacobi as more and more processors are added. In fact, the method is weakly chaotic with the guarantee

that at least one input to each subcircuit in the task queue has a recent iteration value. The results of this implementation are given in Table 7.2 for three different circuits. The table contains the number of CPU-seconds required to perform each simulation as a function of the number of processors used. Note that on a single processor, the Gauss-Seidel algorithm with dynamic ordering is used.

# Proc	decpla (56 nodes)	cramb (149 nodes)	scdac (154 nodes)
1	594.23	1599.32	4041.7
2	325.66	909.63	2138.76
4	181.16	531.18	1222.16
6	135.87	678.51	972.57
8	126.65	399.41	852.59

Table 7.2 : First-Available-Task Approach

The results indicate that reasonable efficiences can be obtained using this approach. The speed-up factors on eight processors are between 4-5 which translates to efficiences of approximately 50-60%. Fig. 7.7 shows the number of processors which are busy during the execution of the DECPLA circuit. It shows that even for the smallest circuit there is enough work to keep all the processors (eight in this case) busy. Note that when the solution at each time point is obtained, the number of busy processors drops to one momentarily to set up the next time point. This is the synchronization at each time point referred to in the previous section.

The next step was to try to improve the performance using a Gauss-Seidel algorithm. Although the Gauss-Seidel algorithm is sequential in nature, there are a number of opportunities for parallelism. Consider, for example, the circuit graph shown in Fig. 7.8(a). The nodes in the graph represent subcircuits and the arcs represent connections

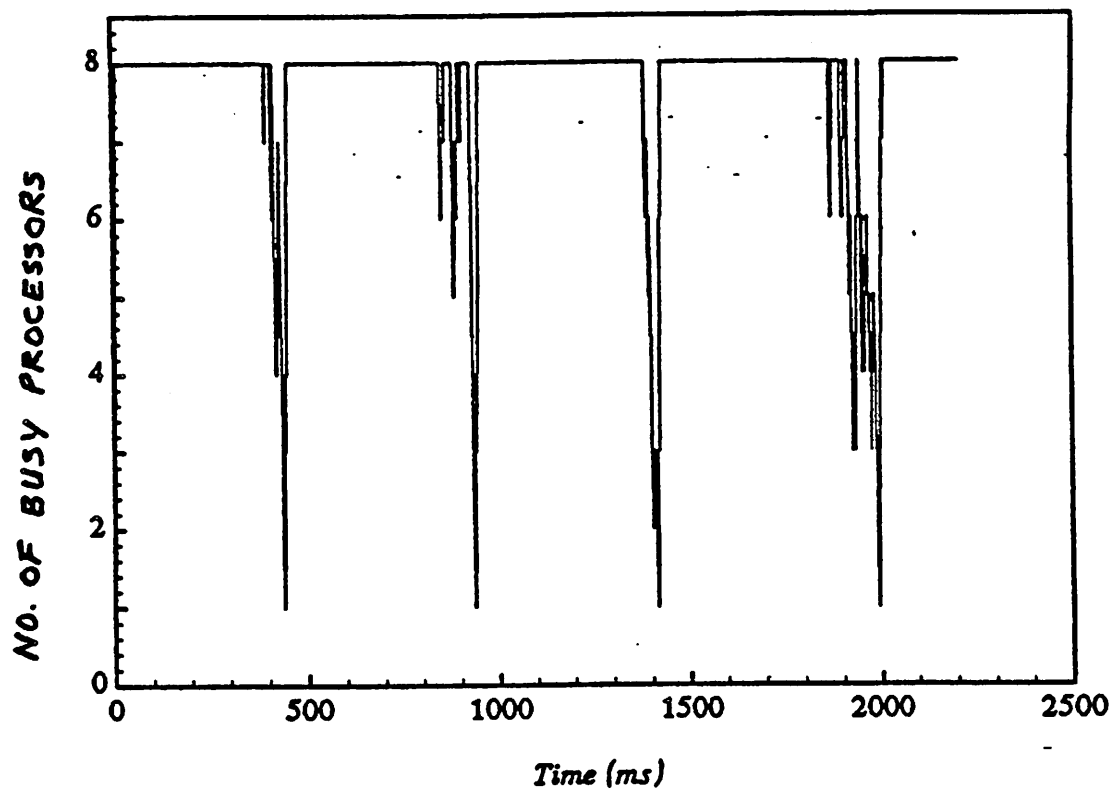


Figure 7.7 : No. of Busy Processor vs. Time for Gauss-Jacobi Method

between the subcircuits. The corresponding task precedence graph is shown in Fig. 7.8(b) for iteration k and iteration $k+1$. In this case, the directed edges between the nodes imply a partial ordering or precedence relation between tasks. Therefore, $S_i < S_j$ 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 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. For example, for the graph in Fig. 7.8(b), the first iteration for S_1, S_2 and S_3 can be computed in parallel since they are all initial tasks. Once these tasks have finished, iteration 1 for S_4 and S_5 can be computed in

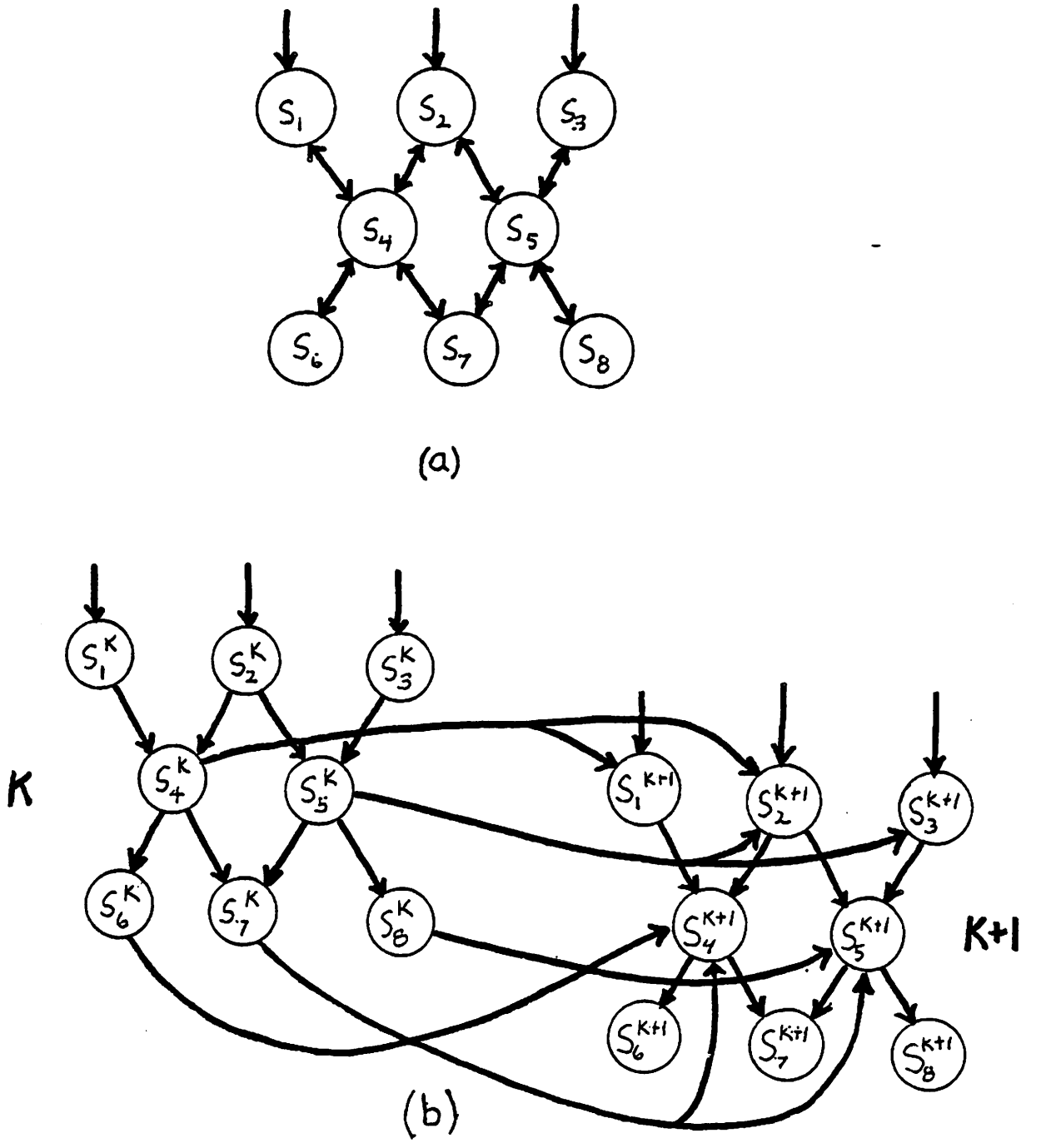


Figure 7.8 : Circuit Graph Example

parallel. After S_4 and S_5 complete execution, two sets of tasks can be computed in parallel: the solutions for S_6, S_7 and S_8 can be computed for iteration 1 and the solutions for S_1, S_2 and S_3 can be computed for iteration 2.

The amount parallelism available using this approach depends on the topology of the circuit. That is, this approach will work quite well on circuits having rather wide graphs but will not be as effective on circuits with narrow graphs. Feedback paths in the circuit also tend to limit the amount of parallelism. In Fig. 7.8(a), if a feedback path existed between S_8 and S_3, then S_3 would have to wait until both S_5 and S_8 finished before beginning its next iteration. Another factor which limits the amount of parallelism is latency exploitation. Some subcircuits may be latent while others may converge quickly during the iterative process so that further processing is unnecessary. However, if the circuit is large enough, the Gauss-Seidel method may be still be efficient. As in the First-Available-Task algorithm, each subcircuit task can also be subdivided into a number of model evaluation subtasks to generate more parallelism. If each subcircuit is large, there may be enough work to keep all p processors busy even if the subcircuits are processed one-at-a-time. Therefore, for a machine with a small number of processors, it may be possible to maintain the strict Gauss-Seidel ordering and obtain a speed-up relative to the uniprocessor version.

The implementation of the Gauss-Seidel method on a multiprocessor requires some complicated checking to determine if a subcircuit task is ready to fire (i.e. ready to be executed). Specifically, the following check is necessary for subcircuit, S_i, to perform iteration $k+1$:

```
if (((S_i active) AND (each fanin is either finished iteration k+1 or latent)
   AND (each fanout is either finished iteration k or latent))                    (7.3)
   OR ((S_i active) AND (S_i is an initial subcircuit performing iteration 1)))
      execute(S_i ):
```

To simplify this check, the subcircuit queue described for the Gauss-Jacobi algorithm can be used. This queue contains all subcircuits that have a change in at least one fanin but the subcircuits may not be ready to fire. Only those subcircuits which satisfy the precedence relations can be processed. The precedence constraints are enforced by searching the queue using the somewhat simpler check:

```
if (((each scheduled fanin has finished iteration k+1)
        AND (each scheduled fanout has finished iteration k))
        OR (S_i is an initial subcircuit performing iteration 1))          (7.4)
  execute(S_i );
```

Note that in the Gauss-Jacobi version, each processor simply picks up a task from the head of the queue. In the Gauss-Seidel case, the queue must be locked and each task checked to see if it is ready to fire using (7.4) above. This increases the time that the queue is locked compared to the Gauss-Jacobi case. The length of time that the queue is held in the locked state depends on the number of tasks in the queue and the likelihood of finding a executable task. If an executable task cannot be found, even though tasks are available in the queue, the processor is said to be *blocked*.

The Gauss-Seidel version was implemented in PSPLICE and the results are presented in Table 7.3.

# Proc	decpla (56 nodes)	cramb (149 nodes)	scdac (154 nodes)
1	621.36	1685.1	4125.92
2	358.72	920.82	2477.22
4	264.51	666.04	1691.33
6	246.92	637.82	1551.99
8	261.27	682.01	2093.47

Table 7.3: strict Gauss-Seidel Version

The results show improvement in the run time up to six processors. Note that the uniprocessor runtimes have increased compared to Table 7.2. This is due to the fact that dynamic ordering is not being used. In this version, the queue is checked and the static Gauss-Seidel ordering is preserved. This checking process introduces additional overhead to the uniprocessor version. Note also that with 8 processors the program actually runs slower due to the blocking problem mentioned earlier and due to the length of time the queue is locked. Even for the largest circuit, for which blocking should be small, the contention for the queue degraded the overall performance. Fig. 7.9 shows the number of busy processors as a function of time for the DECPLA circuit. Comparing it to Fig. 7.7, it is clear that the multiprocessor system is not being fully utilized when it is executing in Gauss-Seidel mode, as the number of busy processors rarely reaches eight. The circuit graph for the DECPLA circuit is given in Fig. 7.10. Note that it is not very wide and there are many feedback paths which limits the degree of parallelism.

Two alternatives exist to improve the performance: either increase the granularity of the computation, or reduce the time that processors are blocked. One way to generate more work is to first try to find a task which is ready to fire, and if none is found, simply take any task from the queue. This task could be from the head of the queue or any other part of the queue. Results from this simple modification are given in Table 7.4. The run times in this case are comparable to the First-Available-Task version, although somewhat slower due to the extra time that each processor has the queue locked. However, the blocking problem has been removed and this accounts for the improvement in performance relative to Table 7.3.

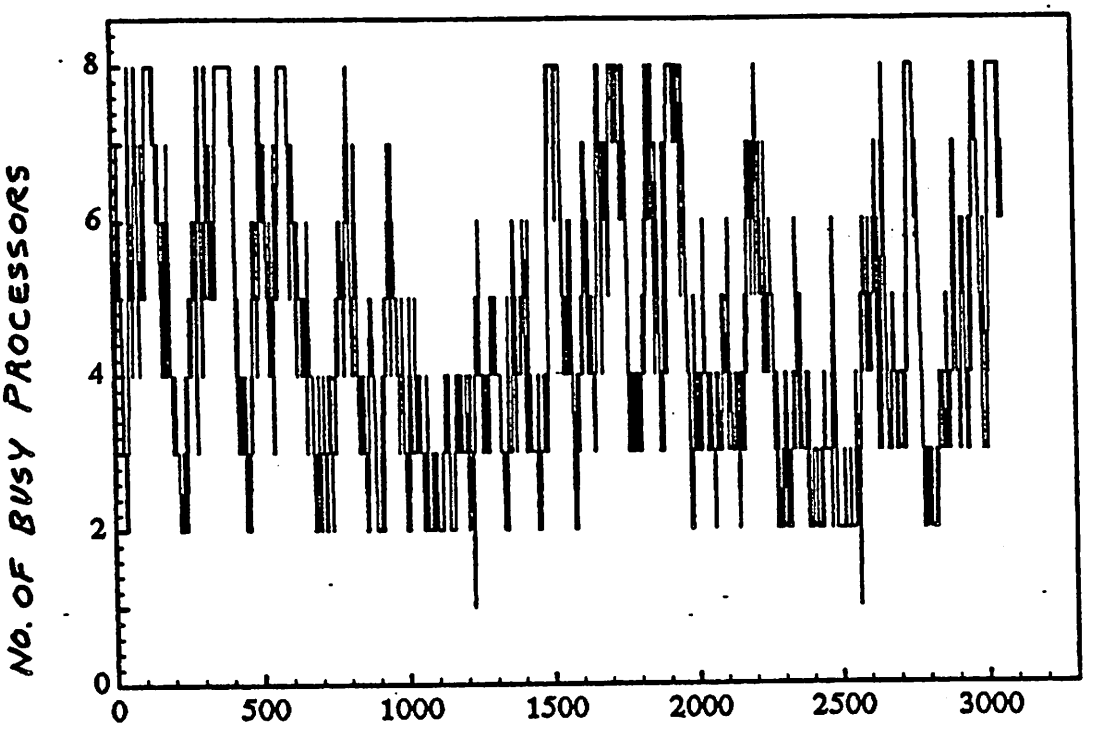


Figure 7.9 : No. of Busy Processors vs. Time for the Gauss-Seidel method

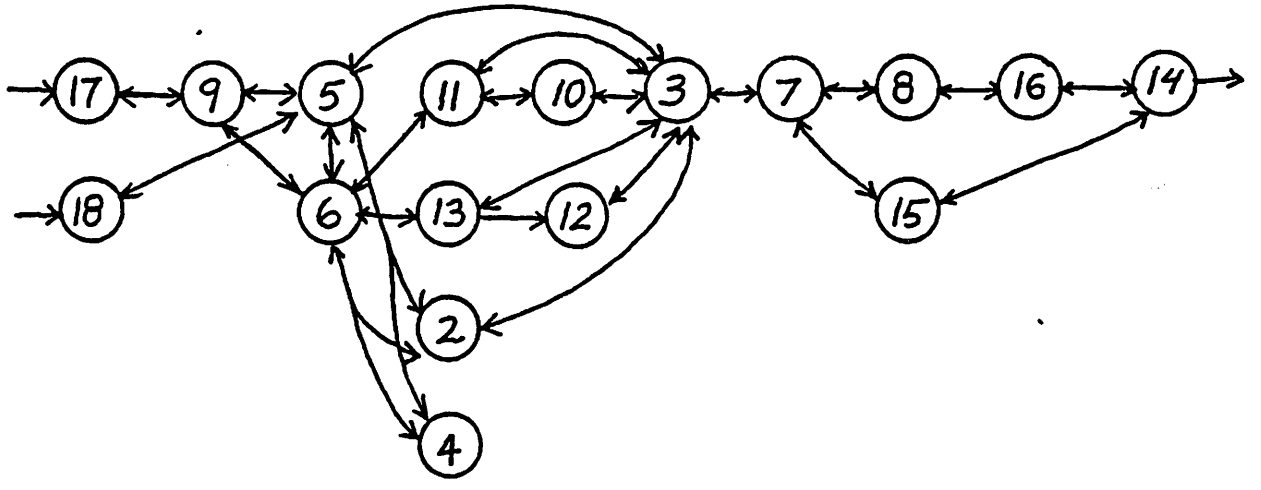


Figure 7.10 : Graph for DECPLA circuit

# Proc	decpla	cramb	scdac
1	596.47	1614.76	4051.49
2	339.4	963.2	2212.49
4	197.52	580.29	1289.49
6	149.29	477.09	1028.24
8	125.83	432.48	910.62

Table 7.4 : Gauss-Seidel/Gauss-Jacobi Version

A few observations can be made based on the limited set of results given here. The First-Available-Task version is the easiest to implement and appears to be the most efficient. The performance of the Gauss-Seidel version depends on the topology of the circuit and is somewhat more complicated to implement. The problem of blocked

processors and the length of time that the queue is locked are limitations of the implementation used here.

7.5. PARALLEL WAVEFORM-NEWTON

The Waveform-Relaxation-Newton (WRN) algorithm was described in Chapter 6 as an extension of ITA to function spaces. It was shown to be faster than both the standard Waveform Relaxation (WR) method and the basic ITA method described in Chapter 4. The WRN method also has features which can be exploited on a parallel processor [Whi85b] and this aspect is examined in this section.

The parallel GS/GJ algorithms described for ITA in the previous sections can be applied to WRN since the considerations for task definition and ordering are similar for both methods. However, based on the results given in [Whi85c], the GJ algorithm may not be as effective for WRN as it was for ITA. In fact, it was shown that the parallel GJ approach was inefficient for the standard WR algorithm in [Whi85b]. Therefore, it is more important to preserve the GS ordering for WRN. The two main problems with preserving the proper ordering in ITA were the time spent in the queue looking for work and the blocking problem when work could not be found. In WRN, each task is much larger than in ITA, involving the solution of each subcircuit over a window interval. This reduces the effect of the queue being locked for a long time. However, it does not remove the blocking problem.

In the parallel implementation of the WR algorithm, a technique called *time-point pipelining* (TPP) [Whi85a] was used to preserve the GS ordering and reduce the blocking problem. This approach can be described using the two-stage inverter chain in Fig. 7.11. Assume that x_1 is assigned to processor 1 and x_2 is assigned to processor 2. In the TPP method, the idea is that processor 2 can start the computation of waveform $x_2^{i}(t)$ before processor 1 finishes the computation of waveform $x_1^{i}(t)$. That is, when

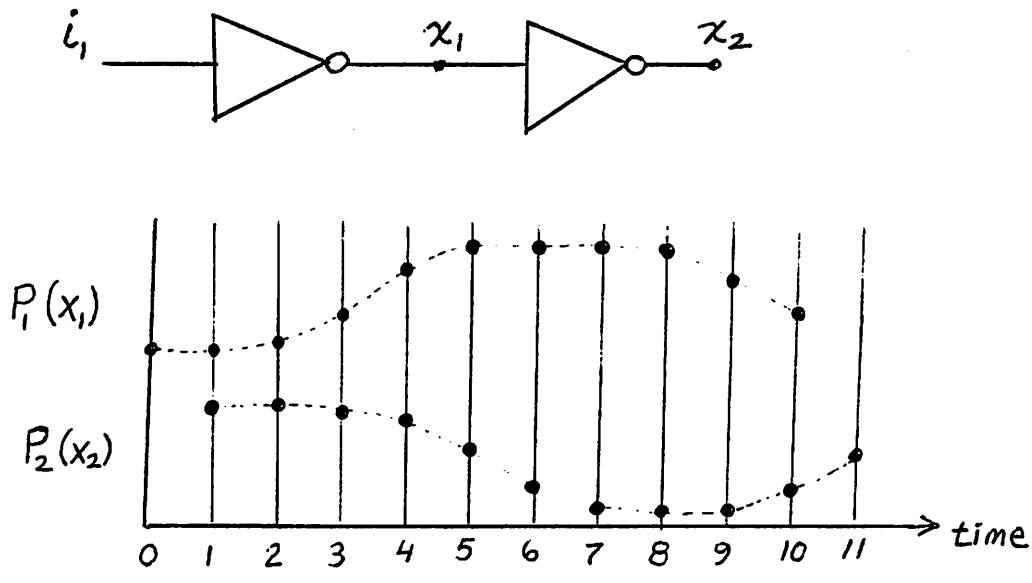


Figure 7.11 : Time Point Pipelining Example

processor 1 computes the first few points of $x_1^{\frac{1}{2}}(t)$, processor 2 can begin to compute the first few points for $x_2^{\frac{1}{2}}(t)$. In this way, the computation of the two waveforms are pipelined.

The TPP approach was shown to be more efficient then the parallel Gauss-Seidel approach for WR [Whi85c] for a number of examples. While the TPP approach could also be used in WRN, the efficiences may not be as high. This is due to the time step control used in WRN as described in Chapter 6. Recall that only one time step is used in the first iteration of WRN, and therefore the notion of time point pipelining is not meaningful. However, as the iterations approach convergence, presumably a large number of points would be used. In this situation, the TPP method would be more effective. One approach to resolve this problem is to use a standard time step control

for WRN when running in parallel, but this is not expected to be as efficient since the extra work in the early iterations may offset the improvement in efficiency.

There is another source of parallelism which may help boost the efficiency in the early iterations. The step refinement strategy given in Chapter 6 implies that the number of time points, m, and their locations are known before the beginning the computation of the waveform for x_i at the k th iteration. Also, the set of waveforms necessary to compute the k th iteration are also known in advance. Therefore, for a given window interval, the Jacobian matrices and portions of the RHS vector can be computed *at all time points in parallel.* This is due to the fact that the Jacobian entries at different time points are independent of each other. Since the location of the time points and the operating point information can be obtained from the waveforms at each time point, the Jacobian matrices can all be computed in parallel. Furthermore, the LU decomposition of the Jacobian matrix at different time points can also be done in parallel. As before, the Jacobian evaluation can be broken down into model evaluation subtasks which can also be done in parallel. The only remaining tasks are to load the RHS entries and to perform the forward-elimination and back-substitution (Fe/Bs) operations for each time point. These tasks cannot be done in parallel across time points. That is, these remaining tasks at a point t_n must be completed before the starting the remaining tasks at t_{n+1}. Therefore, this portion must be done sequentially across the time points. However, this represents only a small fraction of the total time.

The relationship between the various tasks is illustrated in Fig. 7.12 for the inverter chain example of Fig. 7.11. Here, each inverter is considered to be a separate subcircuit. The graph indicates that subcircuit 1 must be started before subcircuit 2, but that the model evaluations for all three time points in the window can be started simultaneously. When the model evaluation for t_1 finishes, the Fe/Bs operation for t_1

can be performed. Note that the Fe/Bs task for t_2 cannot be started until both the model evaluation for t_2 and the Fe/Bs operation for t_1 are completed. A similar restriction exists for Fe/Bs at t_3. If TPP is also used in conjunction with this approach, the model evaluation for subcircuit 2 at time point t_1 can be started after Fe/Bs for subcircuit 1 at time point t_1 is completed. Other such dependencies are denoted using directed arcs. Using this approach, the parallel WRN algorithm is expected to exceed the multiprocessor efficiencies attained by the parallel WR algorithm.

The main drawback of this approach is that it may require a large amount of memory. Normally, the memory required for one matrix is allocated for each subcircuit and this memory is re-used at each time point. However, if the Jacobian matrices at every time point are evaluated in parallel, a separate matrix will be necessary for

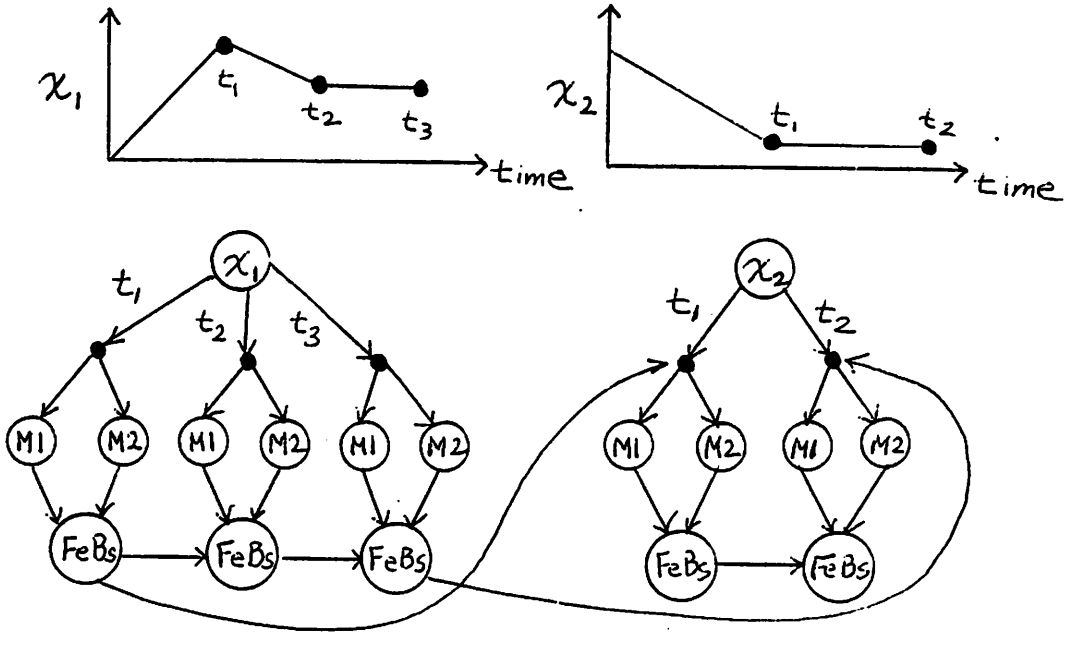


Figure 7.12 : Task Precedence Graph for 2-stage Inverter Chain Example

each time point. This would increase significantly the amount of storage used. If a fast dynamic memory allocation scheme were available, the matrices could be allocated when necessary rather than keeping a separate copy for each time point and each subcircuit. This way, only the matrices for subcircuits being processed would be allocated, thereby reducing the overall memory requirements.

7.6. GENERALIZED SPACE-TIME SCHEDULING MODEL

The relationship between the various parallel schemes presented in the previous sections can be described within the framework of a space-time dependency graph, as depicted in Fig. 7.13. Here, *time* refers to the simulation interval rather than real or physical time. For the purposes of illustration, a fixed time-step grid is used. The "space" axis refers to the position of a node in the circuit, in a conceptual sense. For example, the Gauss-Seidel ordering scheme is one way to specify the spatial ordering of the nodes in a circuit. The node ordering for a three-stage inverter chain, with feedback between adjacent nodes, is shown in the figure. The large dots are time points at which a solution is required and will be referred to as *nodes* of the dependency graph (not to be confused with the circuit nodes). The arcs in the spatial direction imply that the node at the end of an arc is a fanout of the node at the beginning of the arc. The one-way arcs from one time point to the next are due to causality. The causality argument states that events in the past directly affect events in the future, but events in the future have no bearing on events in the past. Therefore, feedback in the time domain is not permitted. The third axis in the figure is the "iteration" axis. Each space-time plane is associated with a particular iteration. Arcs are permitted from a node on iteration plane k to any node on iteration plane j if, and only if, $j \geq k$.

Fig. 7.13 illustrates the decoupled nature of relaxation methods in time and in space. The basic ITA method can be viewed as a spatial decoupling of the circuit nodes.

That is, the circuit nodes are coupled in time but decoupled in space. The WR algorithm allows decoupling in both time and space, since there is no requirement to solve different circuit nodes at the same time. Note that decoupling in time does not imply that the law of causality is violated but simply that different circuit nodes can be solved at different points in time.

The set of nodes and arcs in Fig. 7.13 should be viewed as a task precedence graph for the circuit if a strict Gauss-Seidel ordering is followed and the causality law is obeyed. This task precedence graph is followed closely in the sequential versions of the relaxation-based programs. The nodes in this graph can be scheduled using event-driven techniques, as described earlier. The requirement to begin processing a node is that all nodes associated with incoming arcs be completed first. When a node has been

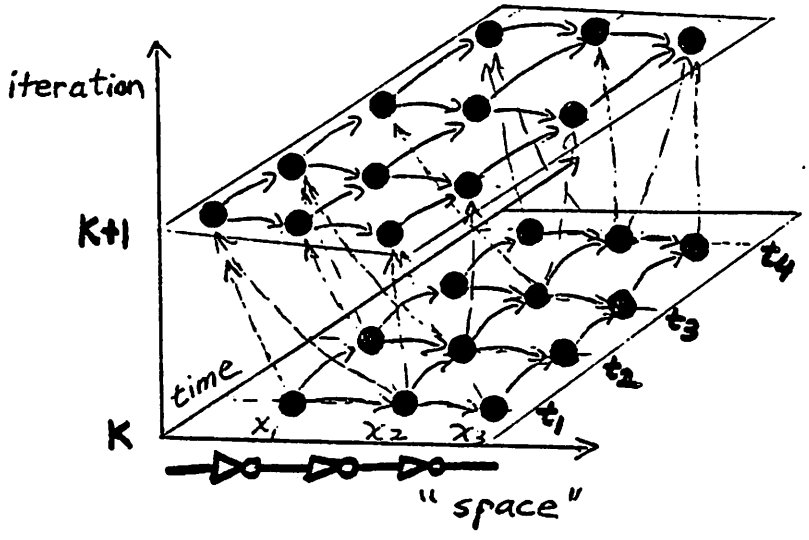


Figure 7.13 : Time-Space Model of Computation

processed, it may schedule any or all of its fanout nodes. In the description to follow, the nodes in the graph will be referenced using (space, time) coordinates for a given iteration k as shown in the figure.

In a parallel implementation, the precedence order given in the graph does not have to be followed strictly and, in fact, some of the algorithms described earlier in this chapter do not adhere to the partial ordering constraints in order to increase parallelism, while others do not take full advantage of the opportunities for parallelism. For example, in the parallel GJ algorithm for ITA, the spatial ordering is violated so that all three circuit nodes can be solved together. However, the implementation in PSPLICE does not take advantage of the fact that events can also be scheduled in the temporal domain. That is, when node (x_1,t_1) is computed, it can schedule both (x_2, t_1) and (x_1, t_2). This approach was described in the section on time point synchronization earlier in this chapter. The timepoint-pipelining approach is a good example of effective exploitation of scheduling in the time domain. In this scheme, when node (x_1,t_1) has computed the solution for iteration k, it schedules both (x_2, t_1) and (x_1, t_2) for iteration k. When these two tasks are completed, they each schedule fanout tasks in time and in space. Note that (x_2,t_2) could also schedule (x_1,t_1) for iteration $k+1$. However, the current implementation of TPP in PRELAX (Parallel RELAX2.3) does not take advantage of this additional opportunity for parallelism [Whi85c]. The chaotic relaxation scheme can be viewed as a scheme which violates the precedence order in both space and iteration, but not time.

The question naturally arises as to whether or not the time point ordering can be violated in any way. The algorithms described for WRN suggests that the nodes (x_1,t_i) for $i=1,2,3,4$ can all computed parallel, and for a GJ version all 12 time points may be computed in parallel! This is not strictly true since the actual solution is obtained by

obeying the causality law. It is only that most of the computation for each time point can be performed in parallel. However, in principle, there is no reason why the time points can not truly be evaluated in parallel. The usefulness of this approach is questionable unless it can be shown that it not "wasted" work. That is, performing an iteration at a future point, before a time point in the past has a new solution, must be shown to be useful. If such an advantage is perceived, then all nodes in a space-time plane can be computed in parallel.

CHAPTER 8

CONCLUSIONS

In this dissertation, a number of new algorithms have been presented to reduce run times for circuit simulation. These algorithms are based on nonlinear relaxation techniques which are extensions of the linear Gauss-Seidel and Gauss-Jacobi iterative methods to nonlinear problems. The new algorithms have been implemented in new programs and used to simulate a variety of industrial circuits. A parallel implementation of one of the algorithms has also been presented. The main results and directions for future work are given below.

Initially, two properties of the waveforms of large digital circuits, called *latency* and *multirate* behavior were defined. Analysis techniques were developed to determine the maximum possible speed improvement that could be obtained by exploiting these two waveform properties. A number of example circuits were analyzed using these techniques and it was concluded that large speed improvements could be obtained if these properties are exploited under ideal conditions. The results indicate that multirate exploitation offers a potentially larger speed improvement compared to latency exploitation. This is because fewer time points are computed in the multirate case. However, actual speed improvement depends on the number of points computed during the simulation and the computational cost of each solution point relative to direct methods. The factors affecting the efficiency of the relaxation-based methods in exploiting the latency and multirate properties were also described. It was shown that the relative cost of each solution point in Waveform Relaxation was higher than the cost for nonlinear relaxation or direct methods.

Two approaches based on Iterated Timing Analysis (ITA) were described, one which exploits the latency property and another which exploits multirate behavior. These approaches were implemented in the SPLICE3 program and a set of industrial circuits were simulated using this program. Both approaches obtained speed improvements relative to the standard simulation approach (i.e, the SPICE2 program) but did not reach the ideal speed improvements. The reasons for this were attributed mainly to an increase in the number of time points used compared to ideal case. This was due to a conservative latency detection scheme (to avoid errors), the use of a conservative time-step control (to avoid rejections) and static partitioning (to improve the convergence speed).

SPLICE3.1, which exploits latency, was implemented using event-driven, selective-trace techniques. In this approach, only the active components in the system are solved at each time point. The active components use a common time-step based on the fastest changing variable in the system. SPLICE3.2 uses an event-driven multirate integration scheme. In this approach, the components are permitted to take different step sizes. Each variable performs two integration operations for each time-step: first, a trial integration to move ahead of the other components and, later, a final integration when the other components catch-up to verify the trial solution. If the waveforms resulting from the trial and final integrations are different, the solution is rejected and a new solution is computed using smaller steps. The rejection is propagated to neighboring components which, in turn, are selectively backed-up and re-integrated in the intervals which are affected by the original rejection. The multirate ITA approach was shown to be faster than the implementation of Waveform Relaxation in RELAX2.3 on a number of examples, especially in the cases where proper window selection was difficult for WR. Future work on this type of algorithm should be to analyze the stability and convergence properties in order to establish robustness of the method. In addition, a

thorough error analysis for the method would be extremely useful.

An algorithm based on extending the nonlinear relaxation methods to function spaces was also described. This algorithm uses a function space Newton method, called "Waveform-Newton" (WN), to solve the Waveform Relaxation (WR) iteration equations. The combined approach is called Waveform Relaxation-Newton (WRN) and can be viewed as a combination of the best features of ITA and WR. The multirate integration is performed using a waveform-based approach and it uses a single WN iteration for each relaxation iteration. This method has been implemented in the SPLAX program using a recursive divide-by-2 time-step control. The simulation results obtained using SPLAX indicate that this method is an effective way to exploit multirate behavior. However, the success of this approach depends on the window sizes used and the nonlinearity of the problem within each window interval. Only a preliminary investigation of this approach has been performed in this dissertation. Future work in this area should focus on the automatic selection of window sizes and on limiting techniques which are best suited to this approach.

Parallel aspects of the ITA and WRN algorithms were also examined in this dissertation. The ITA algorithm of SPLICE3.1 was implemented in the PSPLICE program and a number of issues concerning task granularity, scheduling and ordering were addressed. A novel way to parallelize WRN was described in which most of the computation at different time points in the simulation could be performed in parallel. The parallel algorithms described in this dissertation were presented in the common framework of a generalized space-time-iteration data-flow graph for parallel computation. Much work remains in the area of parallel circuit simulation, especially in extending the methods to a large number of processors. Parallel versions of the WRN and multirate ITA methods should be implemented and compared to the timepoint-pipelining

approach. These methods should also be implemented on other parallel processors to find the best match between algorithms and architecture.

The development of a high-performance, accurate circuit simulation engine is currently an active research area. This type of machine will have a significant impact on the design cycle time and the quality of the designs, since the designer will be able to explore many more design options and be able to thoroughly test the circuit under a variety of operating conditions before it is fabricated. I believe that some of the new algorithms presented here, combined with advanced computer architecture and special-purpose hardware, will provide the necessary performance required for detailed simulation of VLSI circuits.

REFERENCES

[Auc85] B.D.Ackland, S.R. Ahuja, T. L. Lindstron, D.J. Romero, "CEMU - A Concurrent Timing Simulator", *Proc. IEEE Int. Conf. on Computer-Aided Design*, Santa Clara, CA., 1985.

[Bau78] G. Baudet, "Asynchronous Iterative Methods for Multiprocessors", *J. of the ACM*, Vol. 25, No. 2, April 1978, pp. 226-244.

[BBN85] BBN Laboratories, "The Uniform System Approach to Programming the Butterfly Parallel Processor", Version 1, Oct. 1985.

[Bok75] Van Bokhoven, W.M.G., "Linear Implicit Differentiation Formulas of Variable Step and Order", *IEEE Trans. Circ. and Sys.*, CAS-22(2):109-115, Feb. 1975.

[Bok83] W.M.G. Van Bokhoven, "An Activity Controlled Modified Waveform Relaxation Method" *Proc. IEEE Int. Conf. on Circ. and Sys.*, Newport Beach, CA, May 1983.

[Bra72] R.K. Brayton, F.G. Gustavson, G.D. Hachtel, "A New Efficient Algorithm for Solving Differential-Algebraic Systems Using Implicit Backward-Differentiation Formulas", *Proc. IEEE*, Vol. 60, No. 1, pp. 98-108, Jan. 1972.

[Bry80] R. E. Bryant, "An Algorithm for MOS Logic Simulation", *LAMBDA*, 4th Quarter 1980, pp. 46-53.

[Bur83] J. L. Burns, A. R. Newton, D. O. Pederson, "Active Device Table Lookup Models for Circuit Simulation", *Proc. IEEE Int. Conf. on Circ. and Sys.*, Newport Beach, CA, May 1983.

[Car84] C.H. Carlin, A. Vachoux, "On Partitioning for Waveform Relaxation Time-

Domain Analysis of VLSI Circuits", *Proc. Int. Symp. on Circ. and Sys.*, Montreal, Canada, May 1984.

[ChLi75] L. Chua, P. Lin, *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*, Prentice-Hall, 1975.

[Cha75] B.R. Chawla, H.K. Gummel, and P. Kozak, "MOTIS - an MOS timing simulator," *IEEE Trans. Circ. and Sys.*, Vol. 22, pp. 901-909, 1975.

[ChMi71] D. Chazan, W. Miranker, "Chaotic Relaxation", in *Linear Algebra and Its Applications*, Vol.2, 1969, pp. 199-222.

[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 Sys.*, Montreal, Canada, May 1984.

[Coh76] E. Cohen, "SPICE Reference Manual", ERL Memo. No. ERL-M592, June 1976.

[Coh81] E. Cohen," Performance Limits of Integrated Circuit Simulation on a Dedicated Minicomputer System", ERL Memo. No.UCB/ERL M81/29, 22 May 1981.

[Dah63] G. Dahlquist, "A Special Stability Problem for Linear Multistep Methods", *BIT*, 3,pp.27-43, 1963.

[DeM80] G. De Micheli, " New Algorithms for the Timing Analysis of MOS Circuits" Master Report, University of California, Berkeley, 1980.

[DeM81] G. De Micheli, A. Sangiovanni-Vincentelli, "Numerical Properties of Algorithms for the Timing Analysis of MOS VLSI Circuits", University of California, Berkeley, ERL Memo. UCB/ERL M81/25, May 1981.

[DeM83] G. De Micheli, A.R. Newton, A. Sangiovanni-Vincentelli, "Symmetric Displacement Algorithms for the Timing Analysis for VLSI MOS Circuits", *IEEE Trans. on Computer-Aided Design*, Vol CAD-2, No. 3, pp. 167-180, July 1983.

[DeKu69] C.A. Desoer, E.S. Kuh, *Basic Circuit Theory*, McGraw-Hill, 1969.

[Deu84] J.T. Deutsch, A. R. Newton, "A Multiprocessor Implementation of Accurate Electrical Circuit Simulation", *Proc. 19th Design Automation Conference*, Las Vegas, Nv., 1984.

[Deu85] J.T. Deutsch, "Algorithms and Architecture for Multiprocessor-Based Circuit Simulation", Ph.D. dissertation, University of California, Berkeley, ERL Memo. No. UCB/ERL M85/39, May 1985.

[Dum85] D. Dumlugol, J. Cockx, H. De Man, P. Odent, "Segmented Waveform Relaxation Algorithms for Large Scale Circuit Simulation", *Proc. IEEE Int. Symp. on Circ. and Sys.*, Kyoto, Japan, June 1985.

[Ens77] P.H. Enslow, "Multiprocessor Organization - A Survey", *Computing Surveys*, Vol. 9, Mar. 1977.

[Fan77] S. P. Fan, M. Y. Hsueh, A. R. Newton and D. O. Pederson, "MOTIS-C A new circuit simulator for MOS LSI circuits," *Proc. IEEE Int. Symp. on Circ. and Sys.*, April 1977.

[Fie84] G. Fielland, D. Rogers "32--bit Computer System Shares Load Equally Among up to 12 Processors", *Electronic Design*, pp. 153-168, Sept. 1984.

[Fly72] M.J. Flynn, "Some Computer Organization and Their Effectiveness", *IEEE Trans. on Comp.*, C-21, no. 9, Sept. 1972.

[Gau83] M. Guarini and O. A. Palusinski, "Integration of Partitioned Dynamical Systems using Waveform Relaxation and Modified Functional Linearization," *1983 Summer Computer Simulation Proceedings*, Vancouver, Canada, July 1983.

[Gea71] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, N.J., 1971.

[Gea80] C. W. Gear, "Automatic Multirate Methods for Ordinary Differential Equation", *Information Processing 80*, International Federation of Information Processing, 1980.

[Gyu85]. R.S. Gyurscik, "A MOS Transistor Model-Evaluation Attached Processor For Circuit Simulation", Proc. IEEE Int. Conf. on Computer-Aided Design, Santa Clara, CA., Nov. 1985.

[Gyu86] R. Gyurscik, "An Attached Processor for MOS Transistor Model Evaluation", Ph.D. dissertation, University of California, Berkeley, ERL Memo. UCB/ERL M86/82, Oct. 1986.

[Hac71] G.D. Hachtel, R.K. Brayton, F.G. Gustavson, "The Sparse Tableau Approach to Network Analysis and Design", *IEEE Trans. on Circ. Theory*, Vol. CT-18, pp. 101-113, Jan. 1971.

[Hen85] B. Hennion, P. Senn, "ELDO: A New Third Generation Circuit Simulator Using the One-step Relaxation Method" *Proc. IEEE Int. Symp. on Circ. and Sys.*, Kyoto, Japan, June 1985.

[Hil80] D. Hill, "Multi Level Simulator for Computer-Aided Design", Ph.D. dissertation, Dept. of Elec. Eng., Stanford University, 1980.

[Hil81] D. Hillis, "The Connection Machine", Ph.D. dissertation, M.I.T., 1985.

[Ho75] C.W. Ho, A.E. Ruehli, P.A. Brennan, "The Modified Nodal Approach to Network Analysis", *IEEE Trans. on Circ. and Sys.*, Vol. CAS-22, pp. 504-509, June 1975.

[Hsi85] H.Y. Hsieh, A.E. Ruehli, P. Ledak, "Progress on Toggle: A Waveform Relaxation VLSI-MOSFET CAD Program" *Proc. IEEE Int. Symp. on Circ. and Sys.*, Kyoto, Japan, June 1985.

[Hua83] T. Huang, "Analysis of a Method for the Timing Simulation of Large-Scale MOS Circuits Containing Floating Capacitors" Master Report, University of California, Berkeley 1983.

[Jac86] G. K. Jacob, A. R. Newton, D. O. Pederson, "An Empirical Analysis of the Performance of a Multiprocessor-Based Circuit Simulator", *Proc. IEEE Int. Symp. on Circ. and Sys.*, San Jose, CA. 1986.

[Kah75] W.Kahan, "Private notes", 1975.

[Kan59] L. Kantorovich and G. Akilov, "*Functional Analysis in Normed Spaces*", 1959. transl. by D. Brown and A. Robertson, Permagon Press, Oxford, 1964.

[Kat85] R. Katz, S. Eggers, D. Wood, C. L. Perkins, R. Sheldon, "Implementing a Cache Consistency Protocol" *Proc. 12th Annual Int. Symp. on Comp. Arch.* Vol. 13, No. 3, Boston, MA, June 1985.

[Kle83] J.E. Kleckner, R.A. Saleh, A.R. Newton, "Electrical Consistency in Schematic Simulation", *Proc. IEEE Int. Conf. on Circ. and Comp.*, NY, October 1983.

[Kle84] J. E. Kleckner, "Advanced Mixed-Mode Simulation Techniques", Ph.D. dissertation, University of California, Berkeley, May 1984.

[Kuc83] D. J. Kuck, D. Lawrie, R. Cytron, A. Sameh and D. Gajski, "The Architecture and Programming of the CEDAR System", *Proc. LASL Workshop on Vector and Parallel Processing*, Los Alamos, NM, 1983.

[Kun76] H.T. Kung, "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", in **Algorithms and Complexity: New Directions and Recent Results**, J.F. Traub, Ed., Academic Press, New York, 1976.

[Kun85] K.S. Kundert, A. Sangiovanni-Vincentelli, "Nonlinear Circuit Simulation in the Frequency Domain", *Proc. IEEE Int. Conf. on Computer-Aided Design*, Santa Clara, CA., Nov. 1985.

[Kun86] K. S. Kundert, "Sparse Matrix Techniques and their Application to Circuit Simulation", **Circuit Analysis, Simulation and Design**, A.E. Ruehli, ed., North-Holland Pub. Co., 1986.

[Law75] D. H. Lawrie, "Access and Alignment of Data in an Array Processor", *IEEE Trans. on Comp.*, C-24, no. 12, Dec. 1975.

[Lel81] E. Lelarasmee, Private Communication, October, 1981

[Lel82] 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 Sys.*, Vol. 1, n. 3, pp.131-145, July 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.

[New78a] A. R. Newton, D. O. Pederson, "Analysis Time, Accuracy and Memory

Tradeoffs in SPICE2", *Proc. 12th Asilomar Conf. on Circ., Sys. and Comp.*, Asilomar CA, November 1978.

[New78b] A.R. Newton, "The Simulation of Large-Scale Integrated Circuits", Ph.D. dissertation, University of California, Berkeley, ERL Memo. ERL-M78/52, July 1978.

[New79] A. R. Newton, "The Analysis of Floating Capacitors for Timing Simulation," Proc. 13th Asilomar Conf. on Circ., Sys. and Comp., Asilomar CA, November 1979.

[New81] "Timing, Logic and Mixed-Mode Simulation for Large MOS Integrated Circuits", in **Computer-Aids for VLSI Circuit**, Sijthoff & Noordhoff International Publishers, The Hague, pp. 175-239, 1981.

[NeSa83] A.R. Newton, A. Sangiovanni-Vincentelli, "Relaxation-based Circuit Simulation", *IEEE Trans. on Elec. Dev.*, Vol. ED-30, No. 9, pp. 1184-1207, Sept. 1983.

[Ma85] T. Ma, "Partitioning for Relaxation Algorithms", EE219 Project Report, University of California, 1985.

[Mars74] J.E. Marsden, *Elementary Real Analysis*, W.H. Freeman and Company, 1974.

[Mar85] G. Marong and A. Sangiovanni-Vincentelli, "Waveform Relaxation and Dynamic Partitioning for Transient Simulation of Large Scale Bipolar Circuits" *Proc. IEEE Int. Conf. of Computer-Aided Design*, Santa Clara, CA, Nov. 1985.

[Mat85] S. Matisson, "CONCISE, a Concurrent Circuit Simulation Program", Ph.D. dissertation, California Institute of Technology, 1985.

[McC75] W.J. McCalla, "Computer-Aided Circuit Simulation Techniques", to be published.

[Mei78]. Miellou, J.C., "Algorithmes de relaxation a retards", *Revue d'Automatique, Informatique et Researche Operationelle 9*, R-1, April 1975.

[Mok85] M.E. Mokari-Bolhassan, D. Smart, T.N. Trick, "A New Robust Relaxation Technique for VLSI Circuit Simulation", *Proc. IEEE Int. Conf. on Computer-Aided Design*, Santa Clara, CA., 1985.

[OrRh70] J. M. Ortega and W.C Rheinbolt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, 1970.

[Rao85] V. Rao, "Switch-level Timing Simulation of MOS VLSI Circuits", Ph.D. dissertation, University of Illinois, UILU-ENG-85-2207, R-1032, Jan. 1985.

[Ret79] R. Rettberg, C. Wyman, "Development of a Voice Funnel System: Design Report", BBN Report #4098, August 1979.

[Sal82] "Accurate Electrical Simulation in SPLICE1", EE290H Report, University of California, Berkeley, May 1982.

[Sal83] R. A. Saleh, J. E. Kleckner and A. R. Newton, "Iterated Timing Analysis and SPLICE1", *IEEE Int. Conf. on Computer-Aided Design*, Santa Clara, CA., 1983.

[Sal84] R. Saleh, "Iterated Timing Analysis and SPLICE1", Master Report, University of California, Berkeley, 1984.

[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

[Sak81] K.A. Sakallah, "Mixed Simulation of Electronic Integrated Circuits", Ph.D. dissertation, Carnegie-Mellon University, DRC-02-07-81, Nov. 1981.

[Sak85] K. Sakallah. "Polynomial Terminal Equivalent Circuits as Dormant Models in Event Driven Circuit Simulation". *Proc. IEEE Conf. on Computer-Aided Design*, Santa Clara, CA, 1985.

[San79] A. Sangiovanni-Vincentelli, L.K. Chen and L.O. Chua, "A New Tearing Approach-Node Tearing Nodal Analysis", *Proc. IEEE Int. Symp. on Circ. and Sys.*, 1977, pp. 143-147

[Seq84] "BALANCE 8000 Guide to Parallel Programming", Sequent Computers Systems, July 1985.

[ShGo75] L.F. Shampine, M.K. Gordon, "*Computer Solution of Ordinary Differential Equations*" W.H.Freeman Pub., 1975.

[Sei85] C. L. Seitz, "The Cosmic Cube", *Communication of the ACM*, 28:22-33, Jan. 1985.

[Sen82] D. Senderowicz, "An NMOS Integrated Vector-Locked Loop," University of California, Berkeley, Memo. No. UCB/ERL M82/83, 12 Nov. 1982.

[Shi82] T. Shima, T. Sugawara, S. Moriyama, and H. Yamada, "Three-Dimensional Table Look-Up MOSFET Model for Precise Circuit Simulation", *IEEE J. Solid-State Circuits*, vol. SC-17, pp.449-454, June 1982.

[Sto71] Stone, H.S., "Parallel Processing with a Perfect Shuffle", *IEEE Trans. on Comp.*, vol C-20, Feb. 1971.

[SzTh75] S.A.Szygenda and E.W.Thompson, "Digital Logic Simulation in a Time-Based, Table-Driven Environment. Part 1. Design Verification," *IEEE Computer Magazine*, March 1975, pp.24-36.

[Rab79] N.B.G. Rabbat, A. Sangiovanni-Vincentelli and H.Y Hsieh, "A Multilevel Newton Algorithm with Macromodelling and Latency for the Analysis of Large-Scale Nonlinear Circuits in the Time Domain", *IEEE Trans. on Circ. and Sys.*, Vol. CAS-26, pp.733-741, Sep. 1979.

[Var62] R. S. Varga, *Matrix Iterative Analysis*, Prentice-Hall, 1962.

[Vla81] A. Vladimirescu, "LSI Circuit Simulation on Vector Computers", Ph.D. dissertation, ERL Memo No. UCB/ERL M82/75, Oct. 1982.

[War78] D.E. Ward and R.W. Dutton, "A Charge-Oriented Model for MOS Transient Capacitances", *IEEE J. Solid-state Circuits*, vol. SC-13, Oct. 1978.

[Web87] D. Webber, A. Sangiovanni-Vincentelli, "Circuit Simulation on the Connection Machine", *Proc. 22nd Design Automation Conference*, Miami, 1987.

[Wee73] W.T. Weeks, A.J. Jimenez, G.W. Mahoney, D. Mehta, H. Qassemzadeh and T.R. Scott, "Algorithms for ASTAP -- A Network Analysis Program", *IEEE Trans. on Circ. Theory*, Vol. CT-20, pp. 628-234, Nov. 1973.

[Wel82] D.R. Wells, "Multirate Linear Multistep Methods for the Solution of Ordinary Differential Equations", Ph.D. dissertation, Univ. of Illinois, Rep. No. UIUCDCS-R-82-1093.

[Whi83] J. White and A. Sangiovanni-Vincentelli, "RELAX2: A New Waveform Relaxation Approach for the Analysis of LSI MOS Circuits", *Proc. Int. Symp. on Circ. and Sys.*, Newport Beach, May 1983.

[Whi84a] J. White and A. Sangiovanni-Vincentelli, "RELAX2.1 - A Waveform Relaxation Based Circuit Simulation Program" *Proc. Int. Custom Integrated Circuits Conference*

Rochester, New York, June 1984.

[Whi85a] J. White, A.L. Sangiovanni-Vincentelli, "Partitioning Algorithms and Parallel Implementation of Waveform Relaxation Algorithms for Circuit Simulation", *Proc. Int. Symp. on Circ. and Sys.*, Kyoto, Japan, June 1985.

[Whi85b] J. White, R. Saleh, A. Sangiovanni-Vincentelli, A. R. Newton "Accelerating Relaxation Algorithms using Waveform-Newton, Step Refinement and Parallel Techniques" *Proc. Int. Conf. on Computer-Aided Design*, Santa Clara, CA, Nov. 1985.

[Whi85c] J. White, "The Multirate Integration Properties of Waveform Relaxation, with Application to Circuit Simulation and Parallel Computation", Ph.D. dissertation, University of California, Berkeley, ERL Memo. No. UCB/ERL 85/90, Nov. 1985.

[Whi86] J. White, N. Weiner, "Parallelizing Circuit Simulation - A Combined Algorithmic and Specialized Hardware Approach" *Proc. IEEE Int. Conf. Comp. and Design*, NY, Oct. 1985.

[Yan80] P. Yang, "An Investigation of Ordering, Tearing and Latency Algorithms for the Time-Domain Simulation of Large Circuits", Ph.D. dissertation, Report R-891, University of Illinois, Urbana, Aug. 1980.

[Yan80] P. Yang, I.N. Hajj and T.N. Trick, "SLATE: A Circuit Simulation Program with Latency Exploitation and Node Tearing", *Proc. IEEE Int. Conf. on Circ. and Comp.*, October 1980.

[Yan83] P. Yang, B.D. Epler, P.K. Chatterjee, "An Investigation of the Charge Conservation Problem for MOSFET Circuit Simulation", IEEE Journal of Solid-State Circuits, Vol. SC-18, No.1, pp.128-138, Feb. 1983.