proc. Int. Symp. on Circ. and Systems (ISCAS)
June 1988, pp. 623-626, Espoo, Finland. (reformatted)

Latency exploitation in circuit simulation by sparse matrix techniques

J.T.J. van Eijndhoven, M.T. van Stiphout

Eindhoven University of Technology
Dept. of Electrical Engineering
PO box 513, 5600MB Eindhoven, The Netherlands

Abstract

The most important operations for a circuit simulator are component model linearisation, updating the network matrix, performing an L/U decomposition on this matrix, and solving the network variables by forward and backward substitution. Methods are presented to keep ALL these operations localized to the part of the network that is active at the current timepoint, thus obtaining a considerable reduction in computational effort. The methods depend upon the sparse matrix structure itself, yielding a very effective 'fine grained' latency exploitation, contrary to methods based on the large blocks specified by the circuit hierarchy. Results are presented, obtained from an implementation of the algorithms in a piecewise linear circuit simulator with an implicit multirate integration scheme.

Introduction

Traditional circuit simulation programs are known to be extremely computationally expensive. This is mainly due to the basic scheme where at each timepoint a Newton-Raphson iteration is needed, which at every cycle performs a full L/U decomposition and forward backward substitution.

Several attempts were made to use the (designer specified) circuit hierarchy to skip the evaluation of inactive parts of the circuit [2] [3] [8] [9]. These approaches spend some overhead on keeping data integrity on the module boundaries but can be effective for 'good' designs. However designers tend to make little use of hierarchy, creating large modules inefficient for latency exploitation.

As alternative, waveform relaxation is gaining popularity [6]. Here a partitioning in relatively small modules which are analyzed separately, forms an ideal basis for latency exploitation. However the cost of the required repeated analysis and the often used limitation to MOS circuits are serious drawbacks.

The method presented in the next sections is able to limit the computations to the active subnetworks only, by carefully optimizing sparse matrix techniques. Thus skipping operations comes on a 'per matrix element' or 'per vector element' basis, leading to an efficient simulation scheme, independent of the (hierarchical) structuring done by the circuit designer. The method makes use of a rank one (vector product) update strategy on the network matrix, which can be used in a Newton-Raphson based simulator. However we implemented the method in a piecewise linear circuit simulator, where this rank one update comes easily and explicitly available. Furthermore an implicit multirate integration scheme is used (assigning different integration stepsizes to the individual components), leading to an event driven analysis, ideally matching the latency exploitation.

System overview

A DC solution is assumed to be found before the transient analysis starts. The network variables (voltages, currents) are denoted by x. During the transient analysis we will always solve for $\dot{x}$, the time derivative of x, and use the appropriate jacobians and source vectors (J, b) of the components.

For the transient analysis a multirate integration scheme is used. If at time t a component k is assigned its individually optimized timestep h_k, this h_k is used to generate the jacobian (J_k, b_k). This component k now posts an event at time point $\min(t + h_k, t + \Delta t_k)$, where Δt_k is the time distance towards the nearest boundary of the piecewise linear component model. (With given x and $\dot{x}$ this boundary is easily explicitly determined.) If other events (from other components) occure before this $t + h_k$, which affect component k due to modifications in $\dot{x}$, this component k is checked for the validity of its event. If necessary, this event is rescheduled. If the event is finally serviced, the jacobian is allowed to change again. Crossing a boundary of the model corresponds to a rank one update, assigning a new different time step h corresponds to a rank p update, with p the dynamic order of the component (usually one, two or three).

With this scheme, components which are in a relative quiescent state, are automatically assigned large time steps h, and hence generate few events. Nodes in the

network (variables) which are incident to these quiet components only, will show almost no variation in $\dot{x}$, and are probably not affected by other fast changing components. The purpose of this paper is to present methods to exploit this latency behaviour, based upon the sparse matrix structure instead of being based upon a user defined hierarchy.

A new L/U decomposition of the network matrix is now to be determined with every modified jacobian, and a new vector $\dot{x}$ is to be solved. For both of these operations, algorithms will be presented that process only the smallest possible part of the L/U decomposition and the involved vectors.

These algorithms deliver a vector $\Delta\dot{x}$, a sparse update to the vector $\dot{x}$. Now for all nonzero elements (indices) of $\Delta\dot{x}$ the vectors x and $\dot{x}$ are updated to the current time-point, and a new event is scheduled for the components which are incident to these modified elements.

The datastructure for the sparse matrix storage is built with linked C structures. Each nonzero element of the L/U decomposition occupies one structure containing the element value, its row and column indices, and two pointers to the next element in the same row and the next element in the same column. Three arrays are in use with pointers to the first element in each row, the first element in each column, and to each diagonal element (the pivots).

The sparse vectors are also stored with a C structure for each nonzero element. Besides the element value, this structure contains the element index and a pointer to the next structure. The linked datastructures for both the sparse vectors and the sparse matrix are ordered: traversing the list gives the elements in ascending index.

The rank one update

The first operation is to obtain the row and column vectors that define the rank one update to the jacobians(s). For Newton-Raphson based simulators this has to be done inside every Newton iteration, with a scheme as for instance described in [7], section 7.3. In our implementation of an event driven piecewise linear simulator, the crossing of the edge of the model of a component, explicitly gives the required row and column vectors, and no iteration is needed, see [4] or [5].

Assume (A, b) represents the matrix of network equations, with the source-vector b, and a solution $\dot{x}$ of $A\dot{x} + b = 0$ is known. Furthermore the row and column vector (r and c) and scalar p, are known, defining the rank one update to (A, b) for which the L/U decomposition has been determined.
So $(A', b') = (A, b) + c \cdot (r^t, p)$, and $\dot{x}'$ has to be solved from $A'\dot{x}' = b'$.
Let $\dot{x}' = \dot{x} + \Delta\dot{x}$, then substitution leads to:

$$(A + c \cdot r^t)\,\Delta\dot{x} + c \cdot (p + r^t \cdot \dot{x}) = 0$$

In the event driven simulation scheme, the update vectors c and r, correspond to the update of the jacobians of one or a few components, and hence have only a very small number of nonzero entries. Therefor the above source vector $c \cdot (p + r^t \cdot \dot{x})$ will have as few nonzero entries, and can be determined with a corresponding efficiency.

In the next two sections, algorithms are presented to determine the L/U decomposition of $(A + c \cdot r^t)$, and produce the solution $\Delta\dot{x}$ by a forward backward substitution process, where the amount of work is controlled by the number of nonzero updates, and not by the matrix size n.

The L/U decomposition update

The most important step is of course the determination of the new L/U decomposition after the row and column update vectors of the jacobian(s) are found. It is well known that a new L/U decomposition can be efficiently found by an update strategy in $2n^2$ operations (for a full matrix of size n), contrary to the n^3 operations required for an entirely new L/U decomposition. The original algorithm as devised by J.M. Bennett [1] is given below.

Assume a decomposition of A is known according to:

$$A = L \cdot D \cdot U$$

with:

$$L_{ii} = U_{ii} = 1, \quad 1 \le i \le n$$
$$L_{ij} = D_{ij} = D_{ji} = U_{ji} = 0, \quad 1 \le i < j \le n$$

Bennetts algorithm efficiently determines L', D' and U' such that:

$$L' \cdot D' \cdot U' = A' = A + c \cdot r^t$$

The algorithm, simplified for a rank one update only, can be given as:

```
d = 1
for i = 1 to n do
begin   D_ii = D_ii + r_i · d · c_i
        p = c_i · d / D_ii , q = r_i · d / D_ii , d = d - p · D_ii · q
        for j = i + 1 to n do
        begin   c_j = c_j - L_ji · c_i
                L_ji = L_ji + c_j · q
                r_j = r_j - U_ij · r_i
                U_ij = U_ij + r_j · p
        end
end
```

algorithm 1

The Sparse Matrix Implementation

This update strategy is well suited for a sparse matrix implementation, as was pointed out already in [5].

Our contribution is an efficient sparse implementation of the L/U update and forward backward substitution, which is able to skip many pivots, resulting in a very low average computational complexity. This skipping of pivots basically corresponds to skipping part of the circuit, giving the desired latency exploitation. If for instance in the above algorithm $c_i = r_i = 0$ for any i, then the body of the outer for-loop can be entirely skipped.

The vectors c and r are expected to have only a very few nonzero entries. These are stored in sparse form, and the for-loops are modified, generating directly useful indices i without trying all values for i one by one.

The algorithm is given below, in C syntax.

```
void ludec_upd( r, c)
{   /* r and c are the rank one update vectors */
    d = 1.0;
    while (r || c)  /* not both vectors empty yet */
    {   /* get first nonzero elements in either row or col*/
        i = get_first_elmnts( &r, &c, &r_i, &c_i);

        D[i]->value +=  r_i * d * c_i;
        p = c_i * d / D[i]->value;
        q = r_i * d / D[i]->value;
        d -= p * D[i]->value * q;

        /* Update row vector and U-row */
        mutual_update( r, D[i], right, -c_i, q );

        /* Update col vector and L-column */
        mutual_update( c, D[i], down, -r_i, p );
    }
}
```

algorithm 2

In the above program, get_first_elmnts() determines from the first elements in each sparse vector (if not yet empty) the smallest index i of a nonzero element. For this index, the element values r_i and c_i are returned, which are made zero if the index was absent in the respective vector. If a nonzero element of a vector is returned, the vector pointer is set to the next element, ultimately resulting in an empty vector, terminating the enclosing while loop.
In practice it is found that the above routine produces on average only a few indices i to recompute, resulting in a computational complexity much lower than n, the size of the network matrix.

The above algorithm is of course only a basic stripped version: the real program code has to check for the numerical condition of the (new) pivot, and take appropriate measures if necessary.

The mutual_update() routine performs a simultaneous scan over a sparse vector (r or c) and a matrix row or column (U or L). During this scan it generates the required indices j, in the same way as get_first_elements(), performing the updates, due to the inner for-loop of algorithm 1, in a time complexity proportional to the number of nonzero elements encountered. On average these are again only a few elements, almost independent of the matrix (circuit) size.

The forward backward substitution

The algorithm of the previous section generates the L/U decomposition of $(A + c \cdot r^t)$, and now we only need a suitable forward backward substitution to solve for $\Delta\dot{x}$. Again we expect for the source vector $s = c \cdot (p + r^t \cdot xdot)$ only a very small number of nonzero elements, requiring an algorithm which doesn't check all indices one by one. Below the algorithm is given of the forward substitution process.

```
SPARSE_VEC forward_subs( s)
{   /* solve y from Ly+s=0, s is sparse source vector */
    for (y=NULL; s; s = s->next)
    {   j = s->inx;
        vec_fillin( &y, j); /* create new y in reverse order */
        y->value = - s->value;  /* Note Ljj == 1 */
        /* update s[i] by adding Lij * y[j] for i > j */
        mutual_update( s->next, D[j]->next[down],
                                    down, y->value, 0.0);
    } return( y);
}
```

algorithm 3

In the above algorithm, the use of mutual_update() is somewhat overkill (with the given parameters, the matrix column isn't updated), but is suitable as a compact specification of the algorithm. Of course y might have many more nonzero elements than s, depending upon the connectivity and behaviour of the circuit.

The backward substitution process (solving $\Delta\dot{x}$ from $U \cdot \Delta\dot{x} = y$) is comparable and not listed any more. For the backward substitution the matrix elements of U are also scanned column wise, and the y is needed in reverse order (highest indices first), as generated above.

The vector update

The forward backward substitution process yields a sparse vector $\Delta\dot{x}$, for the update of $\dot{x}$ at a certain integration timepoint t. Some elements of x need to be updated now, and a set of components is to be created which are 'active' and have to be checked for their new event.

For a proper sparse update of x, a vector last_update is kept, containing the timepoint of the last update to x. The updating algorithm is now easily stated as:

```
COMPONENT_SET update_x( t, Dxdot)
{     clear_set( &active_components);
      for (; Dxdot; Dxdot = Dxdot->next)
      {     k = Dxdot->inx;
            x[k] += (t - last_update[k]) * xdot[k];
            print( t, k, x[k]);
            xdot[k] += Dxdot[k]; last_update[k] = t;
            add_set( &active_components, fanout(k));
      } return( active_components);
}
```

algorithm 4

For the components in active_components, some connected x and/or $\dot{x}$ are modified and hence a new event for these components is to be determined. If the earliest event is selected, the corresponding rank one update is to be generated, and the entire process repeats.

Experimental results

In the table below, some data is shown on the achieved latency behaviour, and the corresponding locality of the involved computations. For four circuits, the number of circuit components is given, together with the dimension of the network matrix and the number of nonzero elements in the L/U decomposition.

The four circuits are respectively (GCD) a logic circuit built with macromodels of all the logic gates, (PLL) a simple macro-modeled phase locked loop (oscillator) circuit, (shift) a sixteen stage nmos shift register, and (osc11) an eleven-stage ring oscillator built with cmos inverters.

circuit name	# comp	matrix size	L/U elmnts	rank-1 elmnts	F/B elmnts
GCD	602	618	2051	16	14
PLL	9	10	36	10	3.5
shift	160	599	3003	44	634
osc11	22	105	624	50	223

The column 'rank-1 elmnts' gives the average number of elements in the L/U decomposition that was modified after a rank one update to the network matrix. The column 'F/B elmnts' lists the average number of elements in the L/U decomposition that was used (read) by a forward and backward substitution phase. For the latency behaviour, both these numbers are an indication for the amount of CPU effort, and are to be compared with the total number of nonzero elements in the column 'L/U elmnts'.

As can be expected, the mos transistor circuits are more 'transparent' to local variations, have more fill-inns in the matrix, and (thus) more operations affect a larger part of the circuit.

As overall result, the required CPU time for L/U decomposition updates becomes about 1% of the total CPU time in our simulator. The forward backward substitution takes 3% to 9% of the total CPU time. In the shift register example (160 mos transistors and capacitors), the repeatedly computed $\Delta\dot{x}$ contained on average 16 nonzero elements, affecting 6 components.

References

[1] Bennett, J.M.; "Triangular Factors of Modified Matrices", *Numerische Mathematik* vol. 7, pp 217-221, (1965)

[2] DeRosie, J. A.; Roe, P.H.; and Hippel, K.W.; "Hierarchical interconnection of solved subsystems using explicit multiterminal representations", in: *Proc. 13th Int. Symp. on Circuits and Systems*, Houston, Texas, pp. 1015-1019, (1980).

[3] Eijndhoven, J.T.J. van; "A piecewise linear simulator for large scale integrated circuits" PhD Thesis, Eindhoven Univ. of Tech., Eindhoven, The Netherlands, (1984)

[4] Eijndhoven, J.T.J. van; "Piecewise Linear Analysis" in: *Analog Circuits: Computer Aided Analysis and Diagnosis* T. Ozawa (ed.), Marcel Dekker inc., New York, (to appear)

[5] Fujisawa T., Kuh E.S., Ohtsuki T., "A Sparse Matrix Method for Analysis of Piecewise-Linear Resistive Networks", *IEEE Transactions on circuit theory* vol. CT-19, no. 6, pp 571-584, (1972)

[6] Lelarasmee, E.; Ruehli, A.E.; and Sangiovanni-Vincentelli, A.L.; "The waveform relaxation method for time-domain analysis of large scale integrated circuits", *IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst.*, vol. CAD-1, pp. 131-145, (1982).

[7] Ortega J.M., Rheinboldt W.C., *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, (1970)

[8] Rabbat, N.B.G.; Sangiovanni-Vincentelli, A.L.; and Hsieh, H.Y.; "A multilevel newton algorithm with macromodeling and latency for the analysis of large-scale nonlinear circuits in the time domain", *IEEE Trans. Circuits and Syst.*, vol. CAS-26, pp. 733-741, (1979).

[9] Vlach, M.; "LU decomposition and forward-backward substitution of recursive bordered block diagonal matrices", in: *Proc. 16th Int. Symp. on Circuits and Systems*, Newport Beach, Calif., pp. 427-430, (1983).