A Gracefully Degradable VLSI System for Linear Programming

ALAN A. BERTOSSI AND MAURIZIO A. BONUCCCELLI

Abstract—The use of a fault-tolerant VLSI system for storing and solving linear programming problems is presented. The system can bear multiple faults in processing elements and/or links and still function with an acceptable performance degradation. It is based on an interconnection pattern consisting of a complete binary tree in which spare links between cousin nodes are added so as to reconfigure it as a ternary tree. At any given time of a computation, faulty processing elements and/or links are circumvented by using such spare links. We show that the total silicon area required by this structure is only a constant factor higher than that of a complete binary tree. We then use it to give an efficient implementation of the simplex algorithm in which the time required to perform a single pivot step matches a previously established lower bound for tree machines, in spite of faults.

Index Terms—Cousin-connected tree, fault tolerance, gracefully degradable system, linear programming, simplex algorithm, time complexity, VLSI.

I. INTRODUCTION

VLSI technology is expected to allow the construction of chips with hundreds of thousands to millions of gates in the near future. Since the cost of designing such chips is very high and their production yield is very low, the VLSI technique can be effectively used only for large production quantities of the same chip. Thus, only application areas of prominent importance can be realistically addressed in a VLSI setting. Among them are sorting [21], [4], matrix multiplication [16], DFT [17], dictionary and database processing [2], [14], [21], [5], and linear programming [3]. This last application consists of finding a real \((n - 1)\) vector \(x\) (the variable vector) to

\[
\begin{align*}
\text{minimize } & z = cx \\
\text{subject to } & Ax = d \\
& x \geq 0
\end{align*}
\]

(1.1)

where \(A\) is an \((m - 1) \times (n - 1)\) integer matrix (the constraint matrix), \(d\) is an integer \((m - 1)\) vector, and \(c\) is an integer (row) \((n - 1)\) vector (the cost vector).

Linear programming has been envisaged for VLSI realization only very recently. Lower bounds on the time required to perform a single pivot step of the well-known simplex algorithm were given for several interconnection patterns among processing elements (PE's, for short). In particular, it has been shown that trees have much better lower bounds than systolic arrays [3]. Thus, trees represent suitable interconnection patterns for systems implementing the simplex algorithm. In addition, trees have other nice properties from a VLSI viewpoint. Indeed, whenever the so-called “H-layout” is adopted [6], [23], the resulting chip is highly modular and very compact, and requires a few I/O pads (in fact, only the root of the tree communicates with the outside world). This last feature is particularly relevant, since physical limits lower the number of I/O pins on a chip. In practice, it can be hard or even impossible to produce VLSI chips for systems requiring many I/O pads, like systolic arrays.

A problem of paramount importance in VLSI chip design and production is fault tolerance. Indeed, defects in the silicon chip and production errors make PE’s and communication links faulty. This may cause computation fails, in spite of a high number of good devices. In practice, the probability that a chip has faulty devices increases exponentially with its integration density and area [23]. Since faulty devices on a VLSI chip cannot be replaced, the production yield can become unacceptably low. To overcome these drawbacks, fault-tolerance techniques have to be used in the design stage.

The most studied fault-tolerance technique is to hardware configure most of the good devices according to the target interconnection pattern by using extra wires and switches laid out on the chip. As an example, the system Diogenes [19] follows this approach; other examples can be found in [12], [20], [8], [13], [18], and [10]. Unfortunately, if we wish to configure all the good PE’s into a binary tree, the problem is computationally intractable [24].

An alternative technique, instead, is to configure good devices into an irregular interconnection pattern and to design “ad hoc” algorithms that are able to operate under such a changing pattern. Of course, extra links and/or PE’s are required to reduce degradation in performance due to faults. This approach has been usefully adopted for arrays of PE’s [25], [7] and hierarchical network storage systems [9].

In the present paper, this latter fault-tolerance technique is used to provide a gracefully degradable tree-based VLSI system for solving linear programming problems as defined in (1.1). Such a system is based upon an interconnection pattern previously proposed for back-end storage networks [9], although employed in a different manner. We call this pattern cousin-connected tree (CCT, for short). It consists of a complete binary tree in which spare links between cousin nodes are introduced in order to get a graceful degradation of the system performance when faults occur. At any given time of a computation, faulty PE’s and/or links are circumvented by using such spare links. In this way, the system is actually reconfigured into an (incomplete) ternary tree.

The remainder of the paper is subdivided into six sections. Section II contains a formal definition as well as topological properties of CCT’s. In particular, we compare CCT’s with complete binary trees, showing that a CCT with \(N\) unit area nodes and unit width wires can be optimally laid out in a square of \(O(N)\) area. Therefore, CCT’s do not asymptotically require more area than binary trees. We also present experimental results on performance degradation of both CCT’s and binary trees, showing that CCT’s allow on the average a far lower degradation due to faults. In Section III, we sketch the system architecture we will deal with, while in Section IV we define a set of basic procedure schemes. Section V contains some system configuration procedures, while Section VI is devoted to the implementation of the simplex algorithm. All the procedures given in these last two sections are described in terms of the basic schemes previously defined. We show how to perform a single pivot step in \(O(m)\) time, thus attaining the previously proved \(O(m)\) time lower bound for tree machines [3], in spite of faults. Finally, additional remarks and open questions terminate the paper in Section VII.
II. THE INTERCONNECTION PATTERN

The basic interconnection pattern considered in this paper consists of a complete binary tree in which the left (right) son of a generic node \( i \) is directly connected by a link to the left (right) son of the brother of \( i \). The so-connected nodes are called cousins. More formally, a cousin-connected tree (CCT) with \( N = 2^p - 1 \) nodes, indexed \( 1, 2, \cdots, N \), is a graph in which node \( i \) is connected as usual to nodes \( 2i \) and \( 2i + 1 \) (its left and right sons, respectively); moreover, whenever \( [i/2] \) is even and greater than zero, node \( i \) is also connected to node \( i + 2 \) (its cousin). A live CCT is one whose nodes are labeled either faulty, live, or dead as follows:

1) Some nodes in the set \([4, 5, \cdots, N]\) are firstly labeled faulty.
2) A node is successively labeled live if and only if it is reachable from node 1 (the root) via a simple path of nonfaulty nodes each of which is either the father or the cousin of its successor.
3) All the remaining nodes are finally labeled dead.

Of course, a CCT represents a system whose PE’s and communication links correspond to nodes and edges, while a live CCT represents a system with some faulty PE’s (assuming, as usual, that the root and its sons cannot fail [9]). An example is shown in Fig. 1. Note that a node is dead if and only if either its father or cousin (as for node 8) or one ancestor and the brother of the ancestor (as for nodes 28, 29, and 31) are both faulty.

A. Reconfigured Topology

In using a CCT for a computational task, one has to choose a configuration of nonfaulty PE’s into a working interconnection pattern. One choice is to rearrange the live nodes into a tree. Formally, a reconfigured ternary tree (RTT) is a tree which is obtained from a live CCT as follows:

1) Every faulty or dead node is deleted, together with its incident edges.
2) Every edge joining two live cousins both having live fathers is also deleted.

An example is provided in Fig. 2. Of course, if all PE’s of a CCT are nonfaulty, the resulting RTT is a complete binary tree.

Otherwise, it is easy to realize that RTT comes out to be a ternary tree, in which any live node in the CCT having a faulty or dead father becomes an additional son of its cousin.

B. Topological Features

We now give some topological features of CCT’s and RTT’s.

Theorem 2: The degree of a node in a CCT is at most 4.

Both these quantities, being quite low, contribute to the reliability of the system. In particular, nodes whose degree is a low constant are well suitable for VLSI realization.

Recall that the level of node \( i \) in a tree is the number of edges in a simple path from the root to \( i \). We get the following result.

Theorem 3: The level of node \( i \) in an RTT is at most \( 2^l - 1 \), where \( l \) is the level of node \( i \) in the complete binary tree upon which the corresponding live CCT is set up.

Fig. 1. A 31-node live CCT. Faulty, dead, and live nodes are black, shaded, and white, respectively.

Fig. 2. The RTT obtained from the live CCT of Fig. 1. Intercousin links used in the reconfiguration are shown by heavy lines.

Proof: Let node \( i \) have level \( k \) in RTT and let \( j_0, j_1, \cdots, j_k = i \) be the resulting simple path from the root to \( i \). Node \( j_1 \) is the son of the root. Moreover, for \( h = 1, 2, \cdots, k - 1 \), node \( j_h \) is either the father or the cousin of \( j_{h+1} \), in CCT. However, if \( j_{h+1} \) is the cousin of \( j_h \), then \( j_h \) has to be the father of \( j_{h+1} \). Thus, there are in the path at most \((k - 1)/2\) pairs of successive nodes both at the same level in the corresponding binary tree. Hence, \((k - 1)/2 + 1 \leq l \) and so \( k \leq 2l - 1 \).

Since \( l \leq \log_2 N - 1 \) in an \( N \)-node complete binary tree, it is possible in a CCT to reach any live node from the root (and vice versa) by passing through at most \( 2\log N - 3 \) links and \( 2\log N - 4 \) intermediate live nodes. Thus, the communication between any two live PE’s in a system reconfigured as an RTT is \( O(\log N) \), as for complete binary tree.

C. Area Occupancy

We now show that the introduction of extra links between cousins does not asymptotically increase the silicon area required to layout a CCT with respect to a complete binary tree having the same number of nodes.

First of all, let us observe that the terms “left” and “right” reflect a logical organization of the nodes. However, in the VLSI layout of a binary tree, the left and right sons of a node can be physically placed anywhere in the chip.

The layout of a CCT is obtained from that of a complete binary tree as follows. We start with an H-shaped layout for the tree [6], [23], in which the left (right) son of a node which is in turn a right son is physically placed on the right (left) of its father. We then add links between cousins as shown in Fig. 3. Assuming the classical rectangular grid, two-layer model for VLSI layout [23], the following result holds.
connecting them to the root. Moreover, such nodes are connected with the roots of the \( H_{j,2} \)'s by their links for the external world. We add a unit frame around the square, which is used to connect the roots of the four \( H_{j,2} \)'s which become cousins in \( H_j \). Finally, we stretch the square by four horizontal and vertical units, without breaking the existing links, in order to have enough space for the new links between the sons of the roots of the \( H_{j,2} \)'s which become cousins. This completes the layout of \( H_j \), which is inscribed in a square with side \( 2\sqrt{\binom{A_{j-2}}{2} + 7} \). Thus, its area is \( (2\sqrt{\binom{A_{j-2}}{2} + 7})^2 = 4A_{j-2} + 28\sqrt{\binom{A_{j-2}}{2} + 7} + 49 \). Since by assumption \( A_{j-2} = O(2^{j-1}) \), the area required by \( H_j \) is \( O(4\cdot 2^{j-1}) = O(2^{2j-1}) \).

Another good feature of CCT's is modularity. In practice, a larger CCT can be built by grouping together four smaller existing CCT's in a manner similar to that seen in Theorem 4. It is indeed sufficient to lay out intercousin links among the roots by surrounding the four chips, instead of stretching them. Clearly, this does not increase the given area bound.

D. Graceful Performance Degradation

Theorems 3 and 4 tell us that CCT's increase the requirement of area and node-to-node communication time, in presence of failures, only by a small constant factor over those of binary trees. This is a relevant feature, since the system performance degradation in the presence of faults is much lower in CCT's than in binary trees.

To show this, we set up simulation experiments to measure the average performance degradation of both CCT's and binary trees in terms of dead nodes. (Let us define live binary trees in an obvious way, as for CCT's).

We considered pairs of complete binary trees and CCT's both having the same number \( N \) of nodes. We chose three values of \( N \), namely 1023, 32767, and 131071. For each value of \( N \), we divided the simulation in subparts, depending on the value \( F \) of distinct faulty nodes. Five values of \( F \) were chosen, ranging from \( \log N / N \) to \( 4\cdot N \). For each of the 15 subparts, we generated \( F \) distinct integer random numbers, drawn from a uniform distribution between 4 and \( N \). We then deleted the corresponding nodes from both the binary tree and CCT, along with their incident edges. Successively, we labeled the remaining nodes as alive or dead as previously defined. The above procedure was repeated 500 times for each subpart. Notice that in a binary tree, a failure of node \( i \) implies that all the nodes belonging to the subtree rooted at \( i \) are dead. For a CCT, instead, this happens only if also the brother of \( i \) is faulty (e.g., see nodes 14 and 15 of Fig. 1), because of the presence of extra intercousin links. Observe that in both cases dead nodes are not reachable from the root. Thus, they are useless for any computation, since, as in most realistic VLSI systems, the root is the only PE we allow to communicate with the external world. There is only one case in which a reachable node \( i \) is labeled dead, namely, when both its father and cousin are simultaneously faulty (e.g., see nodes 4 and 10 of Fig. 1). In this case, however, all the descendants of \( i \) in CCT still remain alive, whereas they obviously result to be dead in the binary tree.

The simulation programs were written in Pascal and ran on a VAX 11/780 computer. The trial results are summarized in Table I, reporting the average number of dead PE's for each subpart of the experiment. By observing them, it is easy to see that the number of dead nodes in binary trees is much higher than in CCT's. This is remarkable when the number \( F \) of faulty nodes is not too high, say \( \log N \) or \( \sqrt{N} \).

III. SYSTEM ARCHITECTURE

The fault-tolerant system considered in this paper to solve linear programming problems consists of \( N \) processing elements whose interconnection pattern is a CCT. In order to bear several faulty PE's, the system is designed so as to reconfigure its live PE's in an RTT. Such reconfiguration can be trivially done on a local basis, provided each PE knows the status (faulty or not) of its neighbors. We shall not give computational details here. We only observe that four bit flags, say FF, CF, LSF, and RBF, are sufficient for each PE to know the status of its father, cousin, left son, and right son, respectively.
TABLE I  

<table>
<thead>
<tr>
<th>BINARY TREE</th>
<th>CCT</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>58.35</td>
<td>164</td>
<td>303.1</td>
</tr>
<tr>
<td>0.33</td>
<td>5.99</td>
<td>23.04</td>
</tr>
<tr>
<td>500</td>
<td>32</td>
<td>60</td>
</tr>
<tr>
<td>N=1023</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BINARY TREE</th>
<th>CCT</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>182.1</td>
<td>1827</td>
<td>12717</td>
</tr>
<tr>
<td>0.01</td>
<td>11.8</td>
<td>682</td>
</tr>
<tr>
<td>1500</td>
<td>1811</td>
<td>15</td>
</tr>
<tr>
<td>N=32767</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BINARY TREE</th>
<th>CCT</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>174.07</td>
<td>4575.58</td>
<td>48563.3</td>
</tr>
<tr>
<td>0.1</td>
<td>15.67</td>
<td>3570.48</td>
</tr>
<tr>
<td>1700</td>
<td>3631</td>
<td>15</td>
</tr>
<tr>
<td>N=131071</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summarizing, the basic structure of a PE can consist in the above ten flags, along with a few registers, an ALU, and a control unit (CU). By observing the operations that will be described later, one can easily check that only a constant number of registers is required in order to store operation codes, entries of M, row and column indexes, and other values needed for bookkeeping purposes. Registers have a size either equal to b, that of data to be processed, or logN. Operation register codes are only a few bits wide. For the sake of conciseness, we shall not enter into the details of PE architecture in the following. The interested reader can easily come up with a specific architecture after reading the description of the procedures that will follow.

The communication between adjacent PE's is carried out through bidirectional bit-parallel buses. A data bus whose bandwidth b is used to transmit entries of M, while a row and a column bus, both with bandwidth equal to logN, are used to transmit, respectively, row and column indexes of the entries. Finally, there are also an operation code bus and a control bus. This last bus consists of only one bit line which is used to inform whether the associated data have to be taken into account for processing or not.

Lastly, we assume that all PE's operate synchronously, by means of a main clock broadcasting its pulses to every PE. Besides, we assume that the chip is attached to a general purpose host processor. The host processor can directly communicate with the CCT root. It oversees CCT's processing, by initiating new procedures, loading and unloading data, checking partial results, etc. We assume the data size to be that of the host processor word. Typical values for it might be 32 or 64. Thus, throughout this paper, we assume b to be a constant.

Since the size of registers is either a constant or logN, while the bandwidth of buses is either the most logN, the total area required by our system is O(Nlog2N).

IV. BASIC PROCEDURE SCHEMES

We now define six types of basic procedures that can be performed on the proposed system, assuming it is already reconfigured as a RTT. Procedures of the same type follow the same communication scheme among PE's, but require slightly different computations inside the PE's. In this section, we shall stress such communication schemes, leaving the details of the specific computations to the next sections.

Each scheme is initiated by the host processor, which communicates an operation code to the CCT root. This operation code is broadcast to all PE's in RTT via operation code buses. After sending an operation code to its sons, or after receiving it in case of a leaf PE, each PE starts the execution of a suitable program, stored in its CU memory. It keeps executing that program until a new operation code is received. An operation code is often broadcast along with other data, which travel on data, row, and column buses. In particular, row and column indexes are often used as parameters, in the sense that the actions taken by one program may depend on whether or not the incoming indexes match row and/or column indexes stored inside the PE.

The irregularity of the reconfigured topology, due to faults, does not allow a fully synchronous processing. For instance, data coming from different leaves cannot always reach the same PE simultaneously, since they may travel along paths of different length. Thus, data are associated to a control bit, say CB. Whenever CB is 1, the associated data have to be considered for processing, otherwise they will be discarded. PE's store receiving data whose associated CB is equal to 1 and then perform useful computations. In particular, if they expect data to come from the sons, they perform useful computations only when data with CB = 1 have been received from all the sons. Of course, meaningful data are output by a PE only if their CB is set to 1. Otherwise, useless data with CB reset to 0 are output. We will neglect the receiving PE's. This allows us to synchronize operations in the PE's, in spite of the irregularity of the topology.

The six types of procedure schemes are: BROADCAST, SELECT, SENDUP, SENDOWN, OUTPUT, and FLAGSET. We
now sketch them, assuming, for the sake of simplicity, that only meaningful data are received, processed, and transmitted. 

**BROADCAST** is used to broadcast data from the root down to the leaves. Each PE receives data from the father, (permanently or temporarily) stores it in some registers, performs the required computations, and then simultaneously sends the received data to its sons.

The opposite function is carried out by **SELECT**. Each PE waits for data coming from the sons, performs a maximum, minimum, or sum on the received data, and sends the result to its father.

**SENDUP** is analogue to **SELECT**, but each PE receives data from only one son, stores and process them, and successively sends the data up to its father.

With **SENDOWN**, instead, each PE receives from its father data which are used to modify the content of a register, and sends data to just one of its sons.

The purpose of **OUTPUT** is to send up in a pipelined fashion entries stored in different PE's. The scheme is planned in such a way that a sequence of entries is continuously sent by each PE to its father without using buffer memories. To do this, each involved PE uses a register as a timer, and sets it to a proper value upon receiving the operation code. Then, the PE starts sending to its father meaningful data, if any, received from the sons, and decreasing by one its timer at each clock cycle. When the timer becomes zero, the PE sends to its father the entry it is storing. Then, it restarts to send entries coming from the sons.

Lastly, **FLAGSET** has the purpose of setting STORE and COMM flags. Each PE gathers data from the sons and then makes a decision about flag setting. This decision can also cause a command for flag resetting to be sent to one (or more) of its sons.

It is easy to realize that all the above schemes, **OUTPUT** excepted, require an overall time proportional to $T_{op} \log N$ to be executed on RTT, where $T_{op}$ is the time needed by each PE to perform the computation required by a specific operation code. Indeed, according to Theorem 3, the transmission time of data between the root and the other PE's (and vice versa) is $O(\log N)$. Since we assume the linear programming entries to be $b$ bits long, and $b$ is a constant, also $T_{op}$ is a constant. Thus, the above schemes require $O(\log N)$ time. Of course, to output $k$ entries in a pipelined fashion using an **OUTPUT** scheme requires $O(k + \log N)$ time (see, e.g., Section VI for details).

V. PRELIMINARY SYSTEM PROCEDURES

The reconfigured topology of the system is not regular and may change over time whenever failures occur. Therefore, before loading linear programming data, i.e., the $m \times n$ array $M$, we must know whether we have enough space to store $M$ or not. This cannot be deduced by the number of live PE's, since the irregularity of the RTT can cause some PE's to be useless. However, the CCT itself can gather information in a distributed way about the current RTT topology. Depending on the outcome of this gathering, the RTT can in turn be configured as a storing forest and a communication tree, and finally $M$ can be loaded.

A. Gathering Information about RTT

The purpose of this preliminary procedure is to gather information about the current RTT, irrespective of the particular linear programming data to be loaded. Thus, it can be performed only after a CCT reconfiguration took place, because some PE became faulty.

This procedure is performed in two stages. First, the level of each PE in RTT is found out. Second, the number of successors in the subtree rooted at $i$ is determined for each PE $i$ in RTT.

The first stage is initiated by the host processor which gives an appropriate operation code to the root, together with a datum set to zero. A **BROADCAST** scheme is performed, in which each PE stores the incoming value and sends it increased by one to its sons. In this way, letting $T_{bc}$ be the time required by a single PE to carry out the above computation, each PE $i$ will store its level $l_i$ after $l_i T_{bc} + 1$ clock cycles. Since the maximum level $L$ cannot exceed $2 \log N - 3$ (by Theorem 3) and $T_{bc}$ is a constant, all the levels will be computed and stored in $O(\log N)$ time. In the procedures that will follow, the values $L - l_i$ will also be useful. We call such values *colevels*. To evaluate colevel $i$, PE $i$ has to know $L_i$, which is the level of some leaf. Therefore, the host processor communicates a new operation code to the root, at last $(T_{bc} - 1)(2 \log N - 3) + 1$ clock cycles later than the code for the previous operation. The new code is broadcast to all PE's. As soon as a leaf receives the code, it sends its level to its father. A **SELECT** scheme is performed on RTT. Each PE selects the maximum value among those received from its sons, and sends it to its father. After $O(\log N)$ time, the host processor knows $L_i$. Then another **BROADCAST** follows. Each PE receives $L_i$ from its father (assuming, of course, the host processor to be the root's father), subtracts its level to the incoming value, permanently stores the result in a register, and sends $L$ to its sons. The overall time needed is $O(\log N)$.

The second stage is carried out by a **SELECT** scheme. The processing is initiated by the leaves which, upon receiving the proper operation code, store 1 in a register and send it to their fathers. Each PE permanently stores the values coming from its sons in (three) proper registers, which we call **available sons registers** (ASR's). These values are summed up. The result is increased by one and stored in a fourth register, called **AVAILABLE**, whose content is then sent upwards. In this way, PE $i$ has in **AVAILABLE** the number of its successors in the subtree rooted at $i$.

B. CT and SF Determination

Before loading $M$, RTT has to be properly configured into communication tree and storing forest. In particular, $n$ SS's have to be found, each comprising at least $m$ PE's (of course, assuming $mn \leq N$). This is accomplished by setting the flags STORE, COMM, RSST, and VACANCY as follows.

First of all, RTT executes a **BROADCAST** scheme. The value $m$ is broadcast to each PE, which temporarily stores it for future reference. Then a **FLAGSET** scheme follows. As soon as each leaf receives the proper operation code, it sets STORE and VACANCY to 1, and COMM and RSST to 0. Then, it sends the content of **AVAILABLE** up to its father. Each other PE receives the values from the sons and performs the following tests.

1) If all the received values are smaller than $m$, STORE and VACANCY are set to 1, while COMM is set to 0. Besides, if the content of **AVAILABLE** is greater than or equal to $m$, RSST is set to 1; otherwise RSST is set to 0.

2) In contrast, whenever at least one of the received values is greater than or equal to $m$, then COMM is set to 1 and both STORE and RSST to 0 (in this case, VACANCY is useless). Moreover, as observed in Section III, if a value smaller than $m$ is received from a son, the son in question cannot be either in CT or in SF. Then a proper reset code is sent to this son which, as a consequence, will set STORE and COMM to 0.

Since each PE takes constant time for setting its flags, the overall time needed to configure RTT into CT and SF is $O(\log N)$. While loading $M$, however, each PE in CT has to know how many SS's are reachable through its sons and do not store any row of $M$. To do this, a **SELECT** scheme is performed. As soon as a PE whose RSST and STORE flags are both 1 receives the proper operation code, it sends 1 to its father. PE's with RSST = 0 and STORE = 1 do no operation, whereas those with STORE = 0 and COMM = 0 send zeros. In contrast, PE's in CT (i.e., having COMM = 1) permanently store the values coming from the sons in distinct **son vacant registers** (SVR's), sum them up, and send the sum to their fathers. Clearly, if the sum computed by the CCT root is smaller than $n$, the linear programming problem cannot be stored. One can easily realize that also $O(\log N)$ time suffices to execute this procedure on RTT.

C. Data Loading

We now show how linear programming data, i.e., the array $M$, can be loaded and permanently stored in RTT. We assume that the entries of $M$ are represented as $b$ bit numbers (see Section III). Each column
of $M$ will be stored in an SS in such a way that $-z$ and $c_i$, $1 \leq i \leq n - 1$, are maintained in the SS's roots. Moreover, each entry of $M$ will be stored along with its own row and column indexes.

The entries of $M$ are input by the host processor in the RTT root column by column, one entry at a time, starting with column 0 and ending with column $n - 1$. Column $i$ is input starting with the entry in row $m - 1$ and ending with that in row 0. Since each entry is input along with its row and column indexes, row index 0 can be used to mark the end of a column.

During data loading, RTT performs a SENDOWN scheme as follows.

1) PE's with $\text{COMM} = 1$ send the received data to the leftmost son whose SVR stores a positive value. When a row index equal to zero is received, the SVR associated with that son is decreased by one. (We assume the rightmost son to be the cousin, if any.)

2) PE's with $\text{STORE} = 1$ send the received entries to their leftmost son whose ASR stores a positive value, and decrease the content of such register by one. If all the ASR's are zero, the PE itself stores the entry and sets VACANCY to 0.

3) PE's with $\text{SSST} = 1$ (i.e., those being roots of SS's) store the entry received with row index zero, and set VACANCY to 0.

The above procedure allows us to store entries of $M$ in a depth-first manner. Of course, it is also possible to do this in a breadth-first way by simply substituting the rule of data transmission to the leftmost son with that of transmission one son after the other. Anyway, loading one entry takes $O(\log N)$ time. Thus, the overall time required to load all entries of $M$ in a pipelined fashion is $O(mn + \log N)$. An example of RTT storing a $2 \times 2$ linear programming problem is shown in Fig. 5.

VI. THE SIMPLEX ALGORITHM

In this section, we show how the proposed fault-tolerant system can be used to carry out the computation of the well known simplex algorithm. This algorithm is based on the concept of a “basic feasible solution.” A basic solution for the system $Ax = d$ is obtained by solving it for $m - 1$ variables and by setting the remaining $n - m$ ones to zero (of course, we assume $m \leq n$, as usual [15]). The so selected $m - 1$ variables form an ordered set called basis. A basic feasible solution (bfs) is a basic solution in which all variables are nonnegative. The simplex algorithm moves from a bfs to another bfs which differs by one variable so as to improve the objective function $z$. This is accomplished by means of a pivot step. For a detailed description of the algorithm we refer to [15].

In this section, we assume first that a bfs to start with is already known and data reflect this fact. Namely, matrix $A$ contains a permutation of the $(m - 1) \times (m - 1)$ identity matrix and $c$ the reduced costs relative to the current basis [15]. We suppose that the basis is kept by the host processor, e.g., by using a vector whose $i$th entry is the index of the $i$th basic variable. In Section VI-E, we will show how to get a bfs to start with, by means of the classical two-phase method [15].

A. Pivot Column Selection

The first thing to do is to detect the column of the pivot. A classical rule consists in finding the minimum reduced cost. If such a cost is nonnegative, the current bfs is optimal. Otherwise, the desired column has been found.

This rule can be implemented in our RTT via the following SELECT scheme. A proper operation code is broadcast in RTT along with both zero row and column indexes. As soon as a PE with row index equal to zero and column index not equal to zero receives the operation code, it sends a copy of the stored entry to its father along with the associated column index. All the other PE's having $\text{COMM} = 0$ and $\text{STORE} = 0$ send a positive value, say 1. Each PE with $\text{COMM} = 1$ waits for values from its sons, selects the minimum, and sends it to its father, along with the associated column index.

It is easy to convince ourselves that after $O(\log N)$ time the root of

![Fig. 5. The RTT configured as CT and SF storing a $2 \times 2$ linear programming problem. Numbers outside nodes indicate the contents of available registers.](image)

CCT will receive the minimum reduced cost. If it is negative, then the associated column index, say $k$, is stored by the root, and will successively be used to locate the pivot. The host processor takes such index in order to keep track that $x_k$ will enter the basis. Otherwise, no further decrease in the objective function is possible, and the present bfs is optimal. In this latter case, the output stage will follow (see Section VI-D).

B. Pivot Row Selection

Once the pivot column index $k$ has been selected, the pivot row index can then be located as follows. The pivot is that entry $a_{ik}$, if any, such that

$$
\frac{d_i}{a_{ik}} = \min_{1 \leq i \leq n, a_{ik} > 0} \left\{ \frac{d_i}{a_{ik}} \right\}.
$$

The proposed system will find the pivot row index by properly combining an OUTPUT scheme and a BROADCAST one. Row index 0 and column index $k$ are broadcast along with a proper operation code. Upon receiving this code, each PE with mismatching row index and matching column index sets its timer to $\tau + \gamma/2$, where $i$ is its stored row index, and $\gamma$ is its polevel (see Section V). Intuitively, $\gamma$ has the purpose of normalizing the distance from the root to the different PE's, as this would be the same for all of them. It is considered twice since first operation codes have to flow down the tree, and successively entries have to come up. In contrast, $i$ is used to properly sequence these outgoing entries. In this way, entries in column $k$ and their associated row indexes will be output from the proper SS root as a contiguous stream, in increasing row index order, as shown in Fig. 6. As soon as each one of these entries reaches the RTT root, it is broadcast downwards along with column index zero and its own row index. Therefore, PE's in CCT simultaneously send up data coming from their sons, and broadcast down data received from their fathers. Note that this is possible because of the assumed architecture, based on double directional buses: see Section III. PE's whose row and column indexes do not match the incoming ones only broadcast the received entries. In contrast, whenever a PE gets row and column indexes both matching its own indexes, it divides the resident entry by the incoming one, provided that this latter is positive, and temporarily stores the result in a register. If the incoming entry is nonpositive, $-1$ is stored. Since $m$ entries are serially transmitted through the CCT root, after $O(m + \log N)$ time, each PE storing $d_i$ has performed the required division $d_i/a_{ik}$.
Then, a SELECT scheme follows, in which each PE having nonzero row index and zero column index waits for data coming from the sons, computes the minimum among them and the stored entry, but neglecting negative entries, and sends the result to its father, along with the associated row index. If all the involved entries are -1, then -1 is sent up. The PE with row index zero does the same, but considering of course only the entries coming from the sons. Clearly, also PE's with VACANCY = 1, i.e., those storing no entry, will send up -1, while each PE in CT will only transmit the received data upwards.

It is clear that after additional O(log N) time the pivot row index h and the proper value d_h/a_h reach the CCT root. If the received value is -1, then the linear programming problem is unbounded, and the output stage can follow. Otherwise, the host processor takes such row index in order to update the current basis by substituting the hth variable (that leaving the basis) with x_h.

C. Pivoting

Once the pivot a_h has been located, all problem data must be updated so to keep track that we are moving to another bfs. Such an updating is called pivoting and can be performed in two stages. First, each entry in row h of M is divided by the pivot. Thus, each m_{ikj} is set to m_{ikj}/a_h, for all j. Successively, every other entry not in row h is updated as follows: m_{ij} is set to m_{ij} - m_{ik}a_j, for all i and j (i ≠ h) 115.

The first stage is done by a SENDUP scheme, followed by a BROADCAST one. The pivot row and column indexes are broadcast along with the operation code. Thus, the PE storing the pivot will send it to its father, while the remaining PE's will simply transmit upwards the (eventually) received entry. As soon as a_h reaches the CCT root, it is broadcast downwards with row index h. Each PE in the SS whose row index matches the incoming one performs the required division, thus updating its resident entry. This completes the first stage, which therefore takes O(log N) time.

In the second stage, each PE storing entry m_{ij} of M, 0 ≤ i ≤ m - 1, 0 ≤ j ≤ n - 1, i ≠ h, needs two other entries for updating: m_{ik}, which is stored in its own SS, and m_{kj}, which must be received from the SS storing the pivot. In particular, m_{ik} is needed by all but one PE in the SS storing column j of M. This can be done by broadcasting in parallel for all j as follows. A SENDUP scheme is carried out, in which row index h is broadcast, as for the first stage. The only difference is that once the selected entries reach their SS roots, they are temporarily stored there. Then, a BROADCAST scheme follows (using row index h, again). Each SS root broadcasts downwards the entry it previously stored, and then each PE whose row index mismatches the incoming pivot row index h temporarily stores the received entry. When this procedure is over, all (but one) entries of column k have to be distributed all over the RTT. This can be done in a pipelined fashion by combining an OUTPUT and a BROADCAST scheme in a manner very similar to that already used in pivot row selection. The only difference in the OUTPUT scheme is that row index h is broadcast so that a_h is not output. To do this, the involved PE's will set their timers either to i + 2, if 1 ≤ i ≤ h - 1, or to i + 2 + 2 ≤ i, if h + 1 ≤ i ≤ m - 1. In the BROADCAST scheme, instead, each entry travels with its own indexes. When it reaches a PE having the same row index and a different column index, it is temporally stored.

In this way, after O(m + log N) time, each PE permanently storing m_{ik}, i ≠ h, has both m_{ik} and m_{ik} stored in two registers. Then, updating can take place. To do this, a proper operation code is broadcast together with pivot row index h. As soon as a PE with VACANCY = 0 and whose row index mismatches the incoming one receives the operation code, it updates m_{ik} by setting it to m_{ik} - m_{ik}m_{ik}.

A complete pivot step is thus over. A new pivot step can be carried out by repeating the procedures seen above until either an optimal bfs is found, or the problem is discovered to be unbounded.

Since the time for pivoting is dominated by broadcasting entries of column k all over the RTT, it is O(m + log N). Hence, the overall time taken by a complete pivot step is also O(m + log N). This matches the previously established (8m) lower bound for tree machines [3], unless m is o(log N), a very unlikely event.

D. Output of the Optimal Solution

When an optimal solution has been detected, the current values of the basic variables must be output. Of course, nonbasic variables have to be set to zero, while the value of the hth basic variable is d_h (clearly, if the problem is unbounded, nonbasic variable x_h may
assume any positive value, e.g., see [15]. Since the host processor took note of all changes in successive bases, the output of all entries stored in the leftmost SS is enough. This can obviously be accomplished by means of an OUTPUT scheme, in which each PE with column index zero sends its output to the PE at row index 0, and the column index becomes 1. This is repeated for all PEs until the output of all entries is complete. The OUTPUT scheme can be implemented by a simple algorithm, as described in Section VI-C, the (possibly negative) pivot being added to the end of the row before being sent to the next PE.

E. Two-Phase Method

In the simplex algorithm implementation just seen, we assumed that a BFS with was known in advance. Whenever this does not happen, we can find a BFS by means of the two-phase method [15]. The two-phase method consists in solving, by the ordinary simplex algorithm, the following auxiliary problem: find an $m - 1$ real vector $y$ (the artificial variable vector) to

$$\min w = \sum_{i=1}^{m} y_i$$

subject to

$$Ax + by = d$$

$x, y \geq 0$

where $m$ is the $(m - 1) \times (m - 1)$ identity matrix, and $A$ and $d$ are the original problem data of (1.1). Obviously, an initial basis for (6.2) consists of vector $y$. Thus, (6.2) can be loaded as seen in Section V. Of course, the problem size $m$ is now $m \times (m + m)$ and therefore its running time is $O(m^2 + m \log m)$ time, and the system is now ready for loading and solving a new linear programming problem.

F. Time Complexity

Let us now briefly analyze the overall time required by our fault-tolerant VLSI system to solve a linear programming problem.

Loading data either for phase one or two takes $O(mn + m \log N)$ time. Properly computing the reduced index, say $k$, also takes $O(m \log N)$ time. Therefore, performing a single pivot step takes $O(m + m \log N)$ time. The time for the number of steps to solve either phase one or phase two is $O(mn)$ on the average. If there are any artificial variables in the optimal solution then the number grows exponentially in the worst case [15], the overall time for repeatedly pivoting is $O(m^2 + m \log N)$. Assuming $m \geq m_0$, the overall running time results to be $O(mn)$, and is due to data loading, in which we allowed to input only one entry at a time into the CCT.

Since sequential implementations of the simplex algorithm require $O(mn)$ time per pivot step, their average running time is $O(mn)$. Thus, we achieve an $O(m)$ speedup, in spite of faults. Since we use $N \geq mn$ PE's, the resulting average processor utilization is $O(m/N)$.

Whenever it is possible to input several entries at a time, the overall running time can be reduced. For instance, if the CCT is used as an interconnection pattern of a multiprocessor system instead of a VLSI one, we can assume each entry already been stored in the proper PE. Thus, data loading is $O(1)$ and the running time reduces to $O(m)$. In this case, the speedup and average processor utilization become, respectively, $O(n)$ and $O(n/N).

VII. Conclusions

In this paper, we proposed a tree-based fault-tolerant system for solving linear programming problems by means of the well-known simplex algorithm. The system presents several characteristics that make it suitable for practical VLSI (even WSI) realization.

1) It can bear multiple failures in PE's and allow graceful average degradation in performance.

2) It is based on a (slightly modified) complete binary tree, which is one of the most widely used VLSI interconnection patterns.

3) It requires a few I/O pads (indeed, being a modified tree, only one PE communicates with the outside world).

4) It can be compactly laid out in an H-shaped manner with no special (asymptotical) area with respect to binary trees.

5) It is also modular, since smaller systems can be combined together to form a larger one.

6) It performs a single pivot step in $O(m)$ time, thus attaining a previously proved $O(m)$ lower bound for tree machines implementing the simplex algorithm.

7) It achieves an $O(m)$ speedup with respect to the total running time required by sequential implementations of the simplex algorithm.
The proposed system is intended as a contribution in the direction of designing VLSI chips for solving operations research problems. In practice, we should implement several specialized chips spanning a large range of outstanding topics, such as, for instance, finding a maximum flow, a maximum matching, or solving a linear programming problem via Karmarkar's algorithm. Ideally, the purpose should be the setting up of an operations research machine, in which such specialized VLSI chips are attached to a general purpose host computer. Thus far, machines of this kind have already been successfully realized, for instance, for database processing [11].

REFERENCES


A Note on Detecting Sneak Paths in Transistor Networks

S. CHAKRAVARTY AND H. B. HUNT III

Abstract—The problem of detecting sneak paths in transistor networks arises in the minimization of transistor networks [2], [3]. It is shown that the problem of detecting consistent sneak paths in "very simple" transistor networks is co-NP-complete.

Index Terms—Co-NP-complete, l-graphs, minimization of transistor networks, sneak paths.

I. INTRODUCTION

The problem of detecting sneak paths in transistor networks is considered. This problem arises in the minimization of transistor networks [2], [3]. In [2], a polynomial time heuristic for this problem is discussed. The heuristic in [2] is pessimistic in that it often fails to determine that a sneak path does not exist. This leads us to the question: Does a polynomial time algorithm exist for detecting sneak paths? We show that for very simple transistor networks the problem of determining sneak paths is co-NP-complete. Thus, unless P = NP, there does not exist a polynomial time algorithm for determining sneak paths.

Before we can define the notion of sneak paths, we need some definitions. Consider the example in Fig. 1. The pulldown network can be represented by a graph [6] shown in Fig. 2. The graph in Fig. 2 has one source node and one sink node. Every edge is labeled by an input variable or its complement. We will refer to such graphs as l-graphs. It is not difficult to see that l-graphs are a representation of Boolean functions. The l-graph in Fig. 2 represents the complement of the function implemented by the NMOS gate in Fig. 1. If G is an l-graph, then we use f(G) to denote the Boolean function represented by G.

A path in an l-graph is a sequence of edges (SO, n1), (n1, n2), ... (nΛ, SI), where SO and SI are the source and sink nodes, respectively, and ni ≠ nj if i ≠ j. An l-path in an l-graph is a sequence

Manuscript received October 3, 1986; revised October 22, 1987. This work was supported by National Science Foundation Grant DCR-84-03014.

S. Chakravarty is with the Department of Computer Science, the State University of New York, Buffalo, NY 14260.

H. B. Hunt III is with the Department of Computer Science, the State University of New York, Albany, NY 12222.

IEEE Log Number 8825681.