To simplify the sum in the above equation we separate it into two parts: $k \neq j$ and $k = j$. We also note that

$$(\vec{N} - \vec{e}_j)_k = N_k \qquad k \neq j$$

and

$$(\vec{N} - \vec{e}_j)_k = N_k - 1 \qquad k = j.$$

We then obtain

$$L_m(\vec{N} - \vec{e}_j) = \sum_{\substack{k=1 \\ k \neq j}}^{K} N_k \left[\frac{L_{mk}(\vec{N})}{N_k} + D_{mkj}(\vec{N}) \right] + (N_j - 1) \left[\frac{L_{mj}(\vec{N})}{N_j} + D_{mjj}(\vec{N}) \right]. \quad (7)$$

Simple algebraic manipulation results in the following form of the above equation.

$$L_m(\vec{N} - \vec{e}_j) = L_m(\vec{N}) - \frac{L_{mj}(\vec{N})}{N_j} + D'_{mj}(\vec{N}) - D_{mjj}(\vec{N}) \quad (8)$$

where $D'_{mj}(\vec{N}) = \sum_{k=1}^{K} N_k D_{mkj}(\vec{N})$.

Using the same development as above when $\vec{M} = \vec{N} - \vec{e}_c$, we have

$$L_m(\vec{M} - \vec{e}_j) = L_m(\vec{M}) - \frac{L_{mj}(\vec{M})}{(\vec{M})_j} + D'_{mj}(\vec{M}) - D_{mjj}(\vec{M}) - D_{mcj}(\vec{M}) \qquad (\vec{M})_j > 0. \quad (9)$$

Assuming that the values for the $D'_{mj}(\vec{M})\,\forall m, j$, $\vec{M} = \vec{N}$ and $\vec{M} = \vec{N} - \vec{e}_c$ are available *a priori*, then the cost of the Core algorithm is easily seen to be $O(MK)$.

Now consider the computation of the $D'_{mj}(\vec{M})\,\forall m, j$ in the context of the Linearizer algorithm. In each top level iteration of Linearizer, the Core algorithm is called once for population $\vec{M} = \vec{N}$ and for each population $\vec{M} = (\vec{N} - \vec{e}_c)$, $c = 1, 2, \cdots, K$. If, for each of these calls to the Core algorithm, it was required to recompute the D'_{mj} (population vector) then each Core algorithm call would indeed cost $O(MK^2)$. However, in Linearizer

$$D_{mkj}(\vec{N}) = D_{mkj}(\vec{N} - \vec{e}_c) \qquad \forall m, k, j, c \quad (10)$$

and thus

$$D'_{mj}(\vec{N}) = D'_{mj}(\vec{N} - \vec{e}_c) \qquad \forall m, k, j, c. \quad (11)$$

Therefore, we can precompute $D'_{mj}(\vec{N})\,\forall m, j$ at a cost of $O(MK^2)$ and use these values for each of the $K+1$ calls to the Core algorithm. It is simple to see that the cost of Linearizer is then $O(MK^2)$.

In summary, the following modifications are made to the original Linearizer algorithm.

1) In Steps 1 and 5 of the Linearizer algorithm, compute $D'_{mj}(\vec{N})\,\forall m, j$ prior to all other computations and store for use in the calls to the Core algorithm during Step 2 and Step 3.

2) Step 2 of the Core algorithm is replaced by a computation of $L_m(\vec{M} - \vec{e}_j)\,\forall m, j$ using (8) if $\vec{M} = \vec{N}$ or (9) if $\vec{M} = \vec{N} - \vec{e}_c$ with the precomputed values for $D'_{mj}(\vec{N})$ $(= D'_{mj}(\vec{N} - \vec{e}_l))$ and $D_{mkj}(\vec{N})$ $(= D_{mkj}(\vec{N} - \vec{e}_c))$.

IV. Conclusion

We have shown how Linearizer can be reorganized to reduce the computational cost to $O(MK^2)$. This is accomplished without altering the algorithm in any way that affects the results and thus preserves the empirical evidence of the accuracy of the method.

It is tempting to consider the reduction of the space requirements of the Linearizer to $O(MK)$ [from $O(MK^2)$] since we need only values for $D'_{mj}(\vec{N})$ and $D_{mjj}(\vec{N})$. However, each call to the Core algorithm for population $(\vec{N} - \vec{e}_l)$ requires the previous estimates for $L_{mj}(\vec{N} - \vec{e}_l)$. Thus, it does not appear possible to reduce the order of magnitude space requirements for Linearizer without surgery that would materially alter the algorithm.

Acknowledgment

We thank G. Mackintosh for his careful reading and comments on an earlier version of this note.

References

[1] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queueing networks," *J. ACM*, vol. 27, no. 2, pp. 313-322, Apr. 1980.

[2] K. M. Chandy and D. Neuse, "Linearizer: A heuristic algorithm for queueing network models of computing systems," *Commun. ACM*, vol. 25, no. 2, pp. 126-134, Feb. 1982.

[3] E. de Souza e Silva, S. S. Lavenberg, and R. R. Muntz, "A clustering approximation technique for queueing network models with a large number of chains," *IEEE Trans. Comput.*, vol. C-35, no. 5, May 1986.

On the Equivalence of Cost Functions in the Design of Circuits by Costtable

JON T. BUTLER AND KRISS A. SCHUELLER

***Abstract*—In the costtable approach to logic design, a function is realized as a combination of functions from a table. The objective of the synthesis is to find the least cost realization, where realization cost is the sum of the costs of the functions used, plus the cost of combining them.**

The costs of costtable functions are defined by a cost function, which represents chip area, speed, power dissipation, or a combination of these factors. We show that there is an arbitrarily large set S of cost functions all of which yield the same minimal realization from a given costtable. This implies, for example, that every minimal realization of any function over a cost function in S is independent of the actual cost function used. Furthermore, we show that, with *any* cost function, if the cost of combining functions from a costtable F is sufficiently large, the realizations behave as if the cost function belongs to S. That is, any minimal realization of a function f, using costtable F, is *one* of the minimal realizations of f using F and a cost function in S. Our interpretation of these results is that there are not as many distinct costtables as originally thought.

***Index Terms*—Cost function, costtable, logic design, minimization, multiple-valued logic, synthesis.**

I. Introduction

In the costtable approach to the design of logic circuits [1]–[7], a given function is realized by selecting functions from a table and combining them. Associated with each chosen function is a cost. In a sense, all design is done this way. For example, programs are formed

Manuscript received October 15, 1987; revised September 16, 1988. This work was supported by NATO Grant 423/84, by a Chair Professorship tenured at the Naval Postgraduate School, and by NSF Grant MIP-8706553.

J. T. Butler is with the Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA 93943.

K. A. Schueller is with the Department of Mathematics and Computer Science, Youngstown State University, Youngstown, OH 44555.

IEEE Log Number 9034544.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
SEP 1988		

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
On the Equivalence of Cost Functions in the Design of Circuits by CosttaMe	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Electrical and Computer Engineering,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In the costtable approach to logic design, a function is realized as a combination of functions from a table. The objective of the synthesis is to find the least cost realization, where realization cost is the sum of the costs of the functions used, plus the cost of combining them. The costs of costtable functions are defined by a cost function, which represents chip area, speed, power dissipation, or a combination of these factors. We show that there is an arbitrarily large set S of cost functions all of which yield the same minimal realization from a given costtable. This implies, for example, that every minimal realization of any function over a cost function in S is independent of the actual cost function used. Furthermore, we show that, with any cost function, if the cost of combining functions from a costtable F is sufficiently large the realizations behave as if the cost function belongs to S. That is any minimal realization of a functionf, using costtable F, is one of the minimal realizations off using F and a cost function in S. Our interpretation of these results is that there are not as many distinct costtables as originally thought.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**7**	

from a table of instructions with the cost being instruction execution time. In logic design, the table usually consists of functions which are easily designed in the technology used, and the cost can be chip area, power dissipation, speed, etc.

The cost of a realization is the sum of the costs of the component functions plus the cost of combining them. Typically, there is more than one way to realize a given function, and the goal of the design is to find a minimal cost realization. Kerkhoff and Robroek [2] and Robroek [5] introduce the costtable technique for the synthesis of four-valued unary functions implemented in CCD (charge-coupled devices). Their proposed table contains 45 functions, from which all 256 unary functions are synthesized. The cost of each function is an approximation to the chip area occupied by a CCD realization of that function, and the synthesis technique used is exhaustive search. Lee and Butler [3], [4] show a costtable of 24 entries that produces realizations as good as or better than those in [2] and [5]. The proposed synthesis algorithm is still a search; however, nonproductive combinations are eliminated by using the transition count of the function to guide the search. In general, the choice of a costtable is determined by the total cost of the realizations produced; for a given costtable size, one wants a costtable that yields the lowest total cost. Schueller, Tirumalai, and Butler [6] show minimal and near-minimal costtables of all sizes, and from this, find that the costtables of [2] and [3] are not minimal. Also, it is observed that there is a point of diminishing returns with respect to costtable size. That is, while costtables of larger size produce more economical realizations, beyond a certain size, about 10% of the total number of functions to be synthesized, there is little benefit to adding more functions to the costtable. The analysis in [6] is done for five different costs, and it is found that the point of diminishing returns is approximately the same for all costs.

Schueller and Butler [7] show that the "average" costtable is significantly less efficient than the optimal one for small costtables, but very close to the optimal one for large costtables. Since a randomly chosen costtable is likely to be much worse than optimal for small costtable sizes, effective algorithms for finding minimal costtables are important for this case. This applies to all practical applications of costtables, since the number of entries will be small compared to the universe of functions to be realized. In general, it is not easy to find a minimal costtable. However, for the special case of costtables of size one larger than the smallest costtable, a minimal costtable is shown [7]. In addition, it is shown that a search for minimal costtables cannot exclude certain seemingly useless functions, called composite functions that are most efficiently realized by summing other functions.

In this paper, we show that the minimal realization of functions by costtables is relatively unaffected by changes in cost functions or the cost of combining functions (sum, in our case). Costtable realizations are more robust than previously suspected. Specifically, we show that, for any function, *all* minimal realizations under the *linear* cost function are independent of the specific linear cost function used (of which there are infinitely many). We show that, for general cost functions, if the cost of combining costtable functions is sufficiently large, there is a minimal realization of *any* function that is identical to a minimal realization of that function using the linear cost function. We conclude from these results that the understanding of linear cost function is important to the understanding of the costtable synthesis technique.

II. Notation and Introductory Concepts

Let $R = \{0, 1, \cdots, r-1\}$ be a set of r logic values, where $r \geq 2$, and let $X = \{x_1, x_2, \cdots, x_n\}$ be a set of n variables, where x_i takes on values from R. A function $f(X)$ is a mapping $f: R^n \rightarrow R$. An assignment of values to variables in X is represented by a vector v. The value of $f(X)$ for that assignment is $f(v)$. It will be convenient to represent a function by the tuple $\langle a_0, a_1, \cdots, a_{r^n-1} \rangle$, where a_i is the value of f for an assignment of values to v which, interpreted as a base r number, has value i. For example, a four-valued unary function $f(x)$ is represented as the 4-tuple $\langle a_0, a_1, a_2, a_3 \rangle$, where $a_i = f(i)$, for $0 \leq i \leq 3$. Let $U_{n,r}$ be the set of all n-variable r-valued functions.

Let $c(f)$, the *cost* of function f, be a mapping $c: U_{n,r} \rightarrow \boldsymbol{R}$, where $\boldsymbol{R}$ is the set of real numbers. For example, the cost function $c(f)$ used in [2] and [5] correlates closely with the chip area occupied by the most compact implementation of f.

The connecting operation is ordinary vector addition. That is, if f is realized as the sum $f = f_1 + f_2 + \cdots + f_m$, each component of f is the sum of the corresponding components in $f_1, f_2, \cdots, f_m$. If any component sum of the set of functions exceeds $r-1$, the highest logic value, the sum is undefined. Let s be the cost of realizing the vector sum of two functions. Thus, the cost of the realization $f = f_1 + f_2 + \cdots + f_m$ is $c(f_1) + c(f_2) + \cdots c(f_m) + (m-1)s$, where the last term is the cost of $(m-1)$ two-input adders. The two-tuple (c, s) is called a *cost function/sum pair*.

f is a *basis function* if f is 1 for exactly one component and is 0 otherwise. Let BT be the set of all basis functions plus the function with all 0 components. BT is called the *basis costtable*. F is a *costtable* if $BT \subseteq F \subseteq U_{n,r}$. $c_F(f)$, the cost of realizing $f \in U_{n,r}$ with respect to costtable F is

$$c_F(f) = \min_{f_1, f_2, \cdots f_m \in F} \{c(f_1) + c(f_2) + \cdots + c(f_m) + (m-1)s\}$$

where $f = f_1 + f_2 + \cdots + f_m$ and where c is a cost function. The *total cost* $T(F)$ of costtable F is

$$T(F) = \sum_{f \in U_{n,r}} c_F(f).$$

F_t is a *minimal costtable* of size t if $T(F_t) \leq T(F)$, for all F, such that $|F| = t$.

III. Strong Equivalence Between Cost Function/Sum Pairs

We show in this section whole classes of cost function/sum pairs in which the realizations of functions by costtables is independent of the particular cost function/sum pair.

Definition: Let c and d be cost functions and s and t be the corresponding costs of combining functions, respectively. Then, cost function/sum pair (c, s) is *strongly equivalent* to (d, t) iff for any costtable F, and any pair of functions $f_1, f_2 \in U_{n,r}$, $c_F(f_1) < c_F(f_2)$ iff $d_F(f_1 < d_F(f_2)$.

Strong equivalence preserves the relative costs of implementations among functions realized by a costtable. Thus, if two cost function/sum pairs are strongly equivalent, a minimal realization of a function under one pair is a minimal realization under the other. Since strong equivalence is an equivalence relation, it divides the set of all cost function/sum pairs into equivalence classes.

Theorem 1: Every equivalence class induced by strong equivalence contains a cost function/sum pair (c, s), where $s = 0$.

Proof: We show that every cost function/sum pair is strongly equivalent to a cost function/sum pair, where the sum cost is 0. Let (d, t) be an arbitrary cost function/sum pair in some class $\boldsymbol{C}$, and consider pair (c, s), where $c(f) = d(f) + t$, for all $f \in U_{n,r}$, and where $s = 0$. We show that (c, s) is also in $\boldsymbol{C}$. Given an arbitrary costtable F, let a minimum realization of function f in F with respect to (d, t) be $f = f_1 + f_2 + \cdots + f_m$ with cost

$$d_F(f) = d(f_1) + d(f_2) + \cdots + d(f_m) + (m-1)t$$

where $f_i \in F$. Since a minimum realization of f with respect to cost function/sum pair (c, s) costs no more than the realization $f = f_1 + f_2 + \cdots + f_m$, $c_F(f) \leq d_F(f) + t$. Let a minimum realization of f in F with respect to (c, s) be $f = g_1 + g_2 + \cdots + g_p$ with cost

$$c_F(f) = c(g_1) + c(g_2) + \cdots + c(g_p)$$

where $g_i \in F$. Since a minimum realization of f with respect to cost function/sum pair (d, t) costs no more than the realization $f = g_1 + g_2 + \cdots + g_p$, $d_F(f) + t \leq c_F(f)$. Thus, $c_F(t) = d_F(f) + t$. Since $c_F(f)$ and $d_F(f)$ differ only by a constant, (d, t) and (c, s) are strongly equivalent. Q.E.D.

It follows from Theorem 1 that there is no difference in the realizations produced by a costtable where the cost of combining functions

is included in the cost function. That is, if the cost of combining two costtable functions is s and the cost function is $c(f)$, then the minimal realization of any function f is the same as in the case where there is 0 cost in combining two costtable functions and the cost function is $c(f)+s$. Furthermore, given any cost function/sum pair where the sum is 0, there is an arbitrarily large number of cost function/sum pairs which are strongly equivalent to it. In such classes, the cost functions differ by a constant. Next, we show that strong equivalence extends over classes where the cost functions are differentiated by more than a constant.

Definition: Given $f = \langle a_0, a_1, \cdots, a_{r^n-1}\rangle$, the *linear cost* of f is

$$\mathrm{LC}(\langle a_0, a_1, \cdots, a_{r^n-1}\rangle) = k_0a_0 + k_1a_1 + \cdots + k_{r^n-1}a_{r^n-1} + k.$$

For example, consider unary four-valued functions. If $k_i = 1$ and $k = 0$, $\mathrm{LC}(\langle 1122\rangle) = 6$ and $\mathrm{LC}(\langle 2031\rangle) = 6$, and the linear cost is identical to the sum cost discussed in [7]. If $k_i = k = 0$, $\mathrm{LC}(\langle 1122\rangle) = 0$ and $\mathrm{LC}(\langle 2031\rangle) = 0$, and the linear cost is identical to the constant 0 cost discussed in [7].

Theorem 2: All linear cost functions with $s + k > 0$ belong to a single equivalence class induced by strong equivalence, where s is the cost of the sum operation and k is the constant part of the linear cost function.

Proof: Given a costtable F, let a minimal realization of $f \in U_{n,r}$ be

$$f = f_1 + f_2 + \cdots + f_m$$

where $f_i \in F$, which is achieved at cost

$$c_F(f) = c(f_1) + c(f_2) + \cdots + c(f_m) + (m-1)s$$

where c is a linear cost function. The first m terms on the r.h.s. sum to $c(f) + (m-1)k$, where k is the constant term in the linear cost function. Thus,

$$c_F(f) = c(f) + (m-1)(s+k).$$

Since $s + k > 0$, it follows that this minimal realization of f is achieved by summing the least number of functions from F (smallest m). It follows that any linear cost function/sum pair, where $s+k > 0$, is strongly equivalent to any other such cost function/sum pair. Q.E.D.

Theorem 2 shows that there is no difference in the minimal realizations of functions from a costtable between linear cost functions such as the sum cost and the constant 0 cost discussed in [7]. Thus, an algorithm for finding a minimal realization of a function or for finding a minimal costtable applies to all cost functions.

IV. Weak Equivalence Between Cost Function/Sum Pairs

While strong equivalence exists over the restricted class of linear cost functions, we show in this section *any* cost function is related to this class of functions if the cost of the sum operation is large enough.

Definition: Let c and d be cost functions and s and t be the corresponding costs of combining functions, respectively. Then, cost function/sum pair (c, s) is *weakly equivalent* to (d, t) iff for any costtable F and any function $f \in U_{n,r}$, there is a minimal realization of f using (c, s) that is identical to a minimal realization of f using (d, t).

Like strong equivalence, weak equivalence preserves the relative costs of implementations among functions realized by a costtable. However, unlike strong equivalence, it is not necessary that *all* minimal realizations of any function under one cost function/pair be a minimal realization under the other, only that one such minimal realization exists. Since weak equivalence is an equivalence relation, it divides the set of all cost function/sum pairs into equivalence classes.

Lemma 1: For any cost function c, cost function/sum pair (c, s) is weakly equivalent to (LC, t), where LC is a linear cost function, for sufficiently large s.

Proof: When s is sufficiently large, the least cost realization of any function $f \in U_{n,r}$ is a realization requiring the fewest costtable functions. Thus, a minimal realization using cost function/sum pair (c, s) is a minimal realization using cost function/sum pair (LC, t), and so the two cost function pairs are weakly equivalent. The minimal realization of a function f using cost function/pair (c, s) is the lowest cost realization using the minimal number of costtable functions. Q.E.D.

The smallest value of s for which the observation is true is that value which guarantees that there are no realizations of f with more than the minimal number of costtable functions with lower cost than one with the minimal number of costtable functions. As an example of these ideas, consider the area cost function AC [7] and the following costtable

	Function	Cost
1)	⟨0001⟩	4
2)	⟨0010⟩	10
3)	⟨0100⟩	10
4)	⟨1000⟩	7
5)	⟨0033⟩	13
6)	⟨1111⟩	1
7)	⟨3300⟩	20.

The least cost realization of ⟨3333⟩ using three costtable functions is ⟨1111⟩ + ⟨1111⟩ + ⟨1111⟩ at a cost of $3 + 2s$. Using two, the minimal number of costtable functions, the least linear cost realization is ⟨0033⟩ + ⟨3300⟩ at a cost of $33 + s$. Thus, if the latter is to be a minimal realization, $s \geq 33 - 3 = 30$. Thus, 30 is a lower bound on the value of s such that the area cost function/sum pair, (AC, s), is weakly equivalent to (LC, t). This is considerably higher than the sum cost 2 used with the area cost function [2]–[7].

V. Concluding Remarks

The linear cost function is fundamentally important in the costtable approach to the design of logic circuits. We have shown that the realizations by costtable of any function are the same for *any* choice of a linear cost function/sum pair, such that $s+k > 0$. Furthermore, we have shown that the costtable realizations of an arbitrary cost function/sum pair are identical to those produced by a cost function/sum pair where the sum cost is 0.

We have shown that a weaker relationship exists between cost function/sum pairs considering just changes in the sum, the cost of combining functions. That is, *any* cost function/sum pair is weakly equivalent to the linear cost function/sum pair, in the sense that at least one *minimal* realization of any function is the same, for a sufficiently large sum cost.

References

[1] A. El-Barr, Z. G. Vranesic, and S. C. Zaky, "Synthesis of MVL functions for CCD implementations," in *Proc. 16th Int. Symp. Multiple-Valued Logic*, Blacksburg, VA, May 1986, pp. 116–127.

[2] H. G. Kerkhoff and H. A. J. Robroek, "The logic design of multiple-valued logic functions using charge-coupled devices," in *Proc. 12th Int. Symp. Multiple-Valued Logic*, Paris, France, May 1982, pp. 35–44.

[3] J.-K. Lee and J. T. Butler, "Tabular methods for the design of CCD multiple-valued logic," in *Proc. 13th Int. Symp. Multiple-Valued Logic*, Kyoto, Japan, May 1983, pp. 162–170.

[4] J.-K. Lee, "Synthesis techniques for four-valued logic CCD circuits," M.S. thesis, Northwestern Univ., Evanston, IL, Aug. 1982.

[5] H. A. J. Robroek, "The synthesis of MVL-CCD circuits," M.Sc. Rep. 12.3936, Twente Univ. Technol., Enschede, The Netherlands, Dec. 1981.

[6] K. A. Schueller, P. P. Tirumalai, and J. T. Butler, "An analysis of the costtable approach to the design of multiple-valued circuits," in *Proc. 16th Int. Symp. Multiple-Valued Logic*, Blacksburg, VA, May 1986, pp. 42–50.

[7] K. A. Schueller and J. T. Butler, "On the design of cost-tables for realizing multiple-valued circuits," preprint.

[8] P. P. Tirumalai, "Four-valued logic CCD programmable logic arrays," M.S. Thesis, Northwestern Univ., Evanston, IL, June 1984.

Experimental Results on Subgoal Reordering

XUMIN NIE AND DAVID A. PLAISTED

Abstract—**Subgoal reordering is the problem of determining the order for solving a set of subgoals of a goal, in order to improve the efficiency of some search process. In this paper, we report our investigations into the effect of subgoal reordering on the performance of a goal oriented theorem prover, when some simple syntactic heuristics are used to perform subgoal reordering. We show that subgoal reordering using these simple heuristics has a considerable impact on the performance of the prover on a large set of test problems. Some heuristics even provide equally good, and often better, performance in comparison to the hand ordering of the input clauses. The merit of our approach seems to be that we are considering the syntactic aspect of theorem proving. This aspect is simple in form, cheap in its evaluation, and often provides good heuristics, as has been demonstrated by our results.**

Index Terms—**Depth-first iterative deepening search, heuristics, problem reduction format, subgoal reordering, theorem proving.**

I. Introduction

Goal oriented theorem proving systems have some distinctive advantages over some systems based on resolution [1]. In a goal oriented system, a goal is expressed in terms of subgoals and the solutions for a goal are composed of the solutions for its subgoals. One of the advantages of these systems is that it is easy to incorporate heuristic considerations with these systems. One important such consideration is, for example, to detect unachievable goals by a semantics test [7], [2], [10]. Another heuristic consideration is to choose the order in which the subgoals of a goal are solved, since the order in which the subgoals are solved, on one hand, often does not affect the solvability of a goal and, on the other hand, can have a large effect on the efficiency of solving the goal. In this paper, we will discuss our research on this aspect of heuristic consideration in a goal oriented theorem prover.

We will define the terminology first. A *term* is a well-formed expression composed of variables and function symbols. An *atom* is an expression of the form $P(t_1,\cdots,t_n)$ where $t_1,\cdots,t_n$ are terms and P is a predicate symbol. A *literal* is an atom or an atom preceded by a negation sign $\neg$. A literal is *positive* if it is an atom, *negative* if it is an atom preceded by $\neg$. A *clause* is a disjunction of literals. A *Horn-like clause* is of the form L:-$L_1, L_2,\cdots,L_n$, which represents the clause $L \vee \neg L_1 \vee \neg L_2 \cdots \neg L_n$, where L is called the *head literal* and $L_1,\cdots,L_n$ constitute the *clause body*. A general clause C is converted into a Horn-like clause HC as follows. One of the positive literals in C is chosen as the head literal of HC and all other literals in C are negated and put in the clause body of HC. If C contains only negative literals, we use the special literal FALSE as the head literal of HC. A clause only containing negative literals is called a *goal clause*.

Manuscript received October 6, 1987; revised December 5, 1988. This work was supported in part by the National Science Foundation under Grant DCR-8516243 and by the Office of Naval Research under Grant N00014-86-K-0680.

The authors are with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599.

IEEE Log Number 9034545.

A. Modified Problem Reduction Format

The theorem prover we use is an implementation of the modified problem reduction format [6]. We will present the system briefly here to illustrate the structure of the inference system. The modified problem reduction format accepts Horn-like clauses as input. It has an inference rule per input clause plus the *assumption axioms* and the *case analysis rule*. To be specific, assume S is a set of Horn-like clauses. We obtain a set of inference rules from S for the modified problem reduction format as follows. For each Horn-like clause L:-$L_1, L_2,\cdots,L_n$ in S, we have a *clause rule*. We call the Γ's on the left of the arrow $\rightarrow$ the *assumption list*.

Clause Rules:

$$\frac{[\Gamma_0 \rightarrow L_1 \Rightarrow \Gamma_1 \rightarrow L_1],[\Gamma_1 \rightarrow L_2 \Rightarrow \Gamma_2 \rightarrow L_2],\cdots,[\Gamma_{n-1} \rightarrow L_n \Rightarrow \Gamma_n \rightarrow L_n]}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_n \rightarrow L}.$$

We also have the *assumption axioms* and the *case analysis* rule.

Assumption Axioms:

$$\Gamma \rightarrow L \Rightarrow \Gamma \rightarrow L \text{ if } L \in \Gamma \qquad L \text{ is a literal.}$$

$$\Gamma \rightarrow \neg L \Rightarrow \Gamma, \neg L \rightarrow \neg L \qquad L \text{ is positive.}$$

Case Analysis Rule:

$$\frac{[\Gamma_0 \rightarrow L \Rightarrow \Gamma_1, \neg M \rightarrow L], [\Gamma_1, M \rightarrow L \Rightarrow \Gamma_1, M \rightarrow L], |\Gamma_0| \leq |\Gamma_1|}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1 \rightarrow L}.$$

The goal–subgoal structure of the modified problem reduction format is evident, when used in a back chaining manner. Each clause rule, thus each input clause L:-$L_1, L_2,\cdots,L_n$, can be regarded as a decomposition of a goal L into a set of subgoals $L_1, L_2,\cdots,L_n$. The assumption lists are introduced for guaranteeing the completeness of the system and are not of concern to our discussion.

II. Subgoal Reordering

As we have stated, an input clause L:-$L_1, L_2,\cdots,L_n$ decomposes a goal L into a set of subgoals $L_1, L_2,\cdots,L_n$. Furthermore, the subgoals $L_1, L_2,\cdots,L_n$ can be attempted in any order. Our work is based on this observation. We formally state the fact that the subgoals of an input clause can be attempted in any order as follows:

Theorem: The modified problem reduction format is still sound and complete if, for a Horn-like clause L:-$L_1, L_2,\cdots,L_n$, the clause rule is

$$\frac{\text{permute}([L_1, L_2,\cdots,L_n], [M_1, M_2,\cdots,M_n]),\ [\Gamma_0 \rightarrow M_1 \Rightarrow \Gamma_1 \rightarrow M_1], [\Gamma_1 \rightarrow M_2 \Rightarrow \Gamma_2 \rightarrow M_2],\cdots,[\Gamma_{n-1} \rightarrow M_n \Rightarrow \Gamma_n \rightarrow M_n]}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_n \rightarrow L}$$

where permute $([L_1,\cdots,L_n], [M_1,\cdots,M_n])$ produces an arbitrary permutation $M_1,\cdots,M_n$ of $L_1,\cdots,L_n$ each time it is called.

Proof: We note that the soundness and completeness proofs of the modified problem reduction format in [6] do not claim any order of the literals in the clause body each time a clause rule is used. We can conclude that the order produced by permute is correct. □

The theorem implies that we can order the subgoals in a clause rule during the proof process, when the rule is invoked. The theorem prover will require less user guidance if it orders the subgoals automatically during the proof. In the case of logic programming, for instance, the user usually has a very good idea about what the order of the subgoals should be. Thus, ordering subgoals may not be relevant. But in theorem proving, a user may not have the knowledge to specify a good order in the input. To order subgoals automatically can provide a partial answer to this problem. The problem is how to

order the subgoals. We call the process of determining the order of the subgoals in a clause rule during the proof process *subgoal reordering*. There are a couple of issues involved and we will discuss each of them.

The first issue is how to measure the quality of an ordering. It is hard to give a precise quantitative answer in general. We can roughly say that an ordering is good if it can make the search more efficient. To be specific, we can order the subgoals so that the most important subgoal is attempted first or to reduce the branching factors of the search space. To this end, we have defined some *evaluation functions* which measures the "quality" or "importance" of the subgoals and used the values of the evaluation functions to select the ordering. This raises the question about what the evaluation functions should measure. One requirement for the evaluation functions is that the application of them incurs low overhead, since it is going to be a frequent activity to apply the evaluation functions to the subgoals if the subgoals are ordered during the proof. We have considered several evaluation functions.

F1 To evaluate the size of the subgoals where the size of a subgoal is the number of occurrences of predicate symbols, function symbols, and variables.

F2 To evaluate the mass of the subgoals. Given a set of clauses S, the *mass* of a symbol T (predicate or function symbol), denoted by mass(T), is defined to be

$$\text{mass}(T) = \frac{\text{Number of literals in } S}{\text{Number of occurrences of } T \text{ in } S}.$$

For a term t,

$$\text{mass}(t) = \begin{cases} \text{mass}(C) & \text{if } t \text{ is a predicate or function symbol } C \\ 0 & \text{if } t \text{ is variable} \\ \text{mass}(s) & \text{if } t = \neg s \\ \text{mass}(f) + \text{mass}(t_1) + \cdots + \text{mass}(t_n) & \text{if } t = f(t_1, t_2, \cdots, t_n). \end{cases}$$

F3 The number of solutions with the same predicate symbol as a subgoal.

The second issue concerns what algorithm to use to select the ordering for a set of subgoals. Given n subgoals, there are $n!$ possible orderings. An exhaustive search would be too costly and probably not worthwhile, due to the quality of the evaluation functions. We have used a greedy algorithm instead. For a subgoal $\Gamma_0 \rightarrow L$, when a clause rule corresponding to the input clause L:-$L_1, L_2, \cdots, L_n$ is used, the algorithm will be called to determine an ordering among $L_1, L_2, \cdots, L_n$. The algorithm first applies the evaluation function to each of $L_1, L_2, \cdots, L_n$, then sorts them according to their evaluation function values. The resulting order among $L_1, L_2, \cdots, L_n$ will be the order in which they will be attempted. We call this *static reordering* because an ordering among the subgoals will be determined prior to any attempt to solve any subgoal, when a clause rule is invoked. A slight variation to the algorithm leads to the *dynamic reordering*. In dynamic reordering, no order will be determined prior to attempting any subgoal. Rather, each time a subgoal is to be attempted, a subgoal will be selected from the remaining subgoals. To be specific, for any goal $\Gamma_0 \rightarrow L$, whenever a clause rule corresponding to the input clause L:-$L_1, L_2, \cdots, L_n$ is used, a subgoal L_1' among the n subgoals $L_1, L_2, \cdots, L_n$ will be selected and $\Gamma_0 \rightarrow L_1'$ attempted. After $\Gamma_0 \rightarrow L_1'$ returns with $\Gamma_1 \rightarrow L_1'$, another subgoal L_2' will be selected among the remaining $n - 1$ subgoals, etc.

Dynamic reordering can adjust the order based on the progress of the search, such as new variable bindings and newly derived solutions. A problem may arise from the overhead of repeatedly applying the evaluation function. If there are n subgoals, the cost of performing static reordering would be $O(n \log(n))$ and the cost of performing dynamic reordering would be $O(n^2)$ for our algorithm. For short clauses, this would not make a big difference. This seems to be the case for most of our test problems.

We have studied three heuristics for performing subgoal reordering. We note that the order of the subgoals in the input will be preserved if the evaluation functions assign the same value to the subgoals.

H1 *Subgoal having largest size first*. This heuristic is based on several considerations. 1) A larger subgoal usually has a smaller branching factor since the larger size imposes more constraints on unification. 2) A larger subgoal has a more complex structure. This can be regarded as containing more information, thus being more important. 3) In our prover, the solution size contributes to the cost of solving a subgoal. Attempting the larger subgoal first can make the potentially unsuccessful search path stop earlier since larger subgoals will use larger solutions, thus contributing more to the cost.

H2 *Subgoal having the biggest mass first*. This heuristic is used in [10] for the level-subgoal reordering in their prover based on hierarchical deduction. The subgoal with largest mass is likely to contain nonvariable symbols which occur less frequently or to contain more nonvariable symbols. Nonvariable symbols occurring less are more likely to be the symbols in the theorem or the skolem function symbol. Thus, the subgoal with largest mass can be regarded as being the most important. Also a subgoal with large mass is likely to have a small branching factor.

H3 *Subgoal with the least number of solutions with the same predicate symbol as the subgoal first*. The number of solutions with the same predicate symbol as a subgoal does give a bound on the branching factor for this subgoal. In case of a tie, the subgoal with the biggest mass will be first.

III. Related Work

Similar problems are considered in some other goal oriented theorem proving systems [10], [5]. In [10], *level goal reordering* is performed during the proof process where the search process is controlled by suitable selection of the first literal to resolve upon in a goal clause.[1] Its heuristic is to select the literal with the biggest *mass* or with the most complex structure. In SLR-based proof procedures, the choice of the literal can be made dynamically for the application of the extension operation [5]. One heuristic suggested is to select the literal which can be resolved upon with the least number of input chains.

The problem we consider here is similar in nature to the conjunctive problem in [8]. [8] discusses the problem ordering a conjunctive—a set of propositions which share variables and must be satisfied simultaneously—in order to reduce the size of the search. They use the size of the database to estimate the cost of solving a conjunct and determine an ordering of conjuncts which has the least cost by possibly searching through $n!$ possible orderings for n conjuncts. An *adjacency theorem* is proven to cut down the size of the search and some heuristics are also suggested to avoid the search completely. While the basic problem is the same, some assumptions in [8] are not valid in our case. For example, the assumption that all solutions to a conjunct are directly available in the database is not valid. This assumption makes it possible to estimate the cost of solving a conjunct rather easily. In our case, however, the solutions to a subgoal are rarely directly available and require possibly many inferences to obtain; we do not know how many inferences would

[1] Here the term *goal* clause does not refer to an all-negative clause. See [10].

TABLE I
SUBGOAL REORDERING USING H1

Theorem	No Reordering		Dynamic Reordering		Static Reordering	
	seconds	inferences	seconds	inferences	seconds	infereces
dbabhp	6.15	151	6.50	167	6.82	167
fex4t1	590.18	2858	1635.57	3755	1637.50	3755
fex4t2	126.02	1015	126.07	1015	132.00	1015
fex5	231.78	2970	268.83	3710	269.07	3708
ls108	1557.38	8380	29.37	575	29.25	575
ls65	164.20	2957	140.72	2703	146.07	2703
schubert	748.45	8921	649.52	11014	761.42	11014
wos1	64.22	1059	47.32	922	53.12	974
wos10	223.05	3950	200.65	3615	238.52	3950
wos11	235.53	4222	224.12	4070	247.23	4166
wos15	6045.23	20556	62.38	1403	66.80	1458
wos31	7742.22	29783	21079.78	78273	21162.65	78274

TABLE II
SUBGOAL REORDERING USING H2

Theorem	No Reordering		Dynamic Reordering		Static Reordering	
	seconds	inferences	seconds	inferences	seconds	infereces
dbabhp	6.15	151	40.82	598	41.55	598
fex4t1	590.18	2858	1670.18	3755	1681.22	3755
fex4t2	126.02	1015	126.98	1015	129.27	1015
fex5	231.78	2970	259.57	3644	266.15	3632
ls108	1557.38	8380	29.43	575	30.37	575
ls65	164.20	2957	100.37	1786	102.78	1786
schubert	748.45	8921	2306.73	24208	695.25	12680
wos1	64.22	1059	681.03	5665	696.50	5680
wos10	223.05	3950	136.68	2282	138.40	2322
wos11	235.53	4222	239.52	3732	245.40	3905
wos15	6045.23	20556	62.08	1339	61.57	1382
wos31	7742.22	29783	5816.27	28086	5987.40	28136

TABLE III
SUBGOAL REORDERING USING H3

Theorem	No Reordering		Dynamic Reordering		Static Reordering	
	seconds	inferences	seconds	inferences	seconds	infereces
dbabhp	6.15	151	41.78	598	41.72	598
fex4t1	590.18	2858	603.50	2901	612.58	2901
fex4t2	126.02	1015	146.33	1017	146.48	1017
fex5	231.78	2970	217.17	2712	215.95	2712
ls108	1557.38	8380	28.93	541	30.20	541
ls65	164.20	2957	101.82	1793	102.32	1793
wos1	64.22	1059	3501.13	12334	3457.18	12354
wos10	223.05	3950	176.95	2788	154.53	2666
wos11	235.53	4222	274.10	3938	266.88	4088
wos15	6045.23	20556	4109.68	14304	3953.87	14528
wos31	7742.22	29783	7141.10	28634	7090.12	28648
schubert	748.45	8921	435.52	9510	712.82	11263

S1: Average Data for Subgoal Reordering			
	Average Time Per Theorem	Average Inference Per Theorem	Average Inference Per Second
no reordering	232.05	1335.76	5.76
dynamic-H1	315.04	1648.72	5.23
static-H1	318.68	1649.38	5.18
dynamic-H2	154.44	1175.12	7.61
static-H2	168.82	1171.12	6.93
dynamic-H3	252.62	1361.39	5.38
static-H3	253.65	1407.85	5.55

S2: Problem Distribution in Running Time (in seconds)						
	(0, 10]	(10, 60]	(60, 300]	(300, 600]	(600, +∞)	Total
no reordering	52	16	8	1	5	82
dynamic-H1	51	21	7	0	3	82
static-H1	52	20	7	0	3	82
dynamic-H2	50	21	7	1	3	82
static-H2	50	20	6	1	5	82
dynamic-H3	50	20	6	1	5	82
static-H3	50	20	6	0	6	82

S3: Comparing with no Reordering					
functions	improvements[3]		degeneration[3]		Even[3]
	number	average(%)	number	average(%)	number
dynamic-H1	33	15.73	20	53.4	29
static-H1	24	19.8	17	31.9	41
dynamic-H2	45	24.9	20	84.0	17
static-H2	38	24.6	24	108.0	20
dynamic-H3	45	22.5	20	164.3	17
static-H3	40	21.2	25	136.9	17

be required. Also, the cost of solving the same subgoal may vary if caching is performed, where caching implies that a subgoal need not be solved more than once. All these make realistically estimating the cost of solving a subgoal very difficult. It is also pointed out in [8], in additon to the difficulty of estimating cost, an optimal ordering of the conjuncts cannot always be achieved by only considering the subgoals of a goal if inferences are required to obtain the solutions. This implies that a global data structure is needed to store all the unsolved subgoals and the optimal ordering is selected from all the possible orderings of those unsolved subgoals.

In our work, instead of estimating the cost of solving a subgoal, we quantify certain syntactic characteristics of the subgoals and use a cheap greedy algorithm to determine the ordering. We only deal with subgoals belonging to one goal to make the subgoal reordering process compatible with the depth-first iterative deepening search [4] used in the prover. The major advantages of the depth-first iterative deepening search are that it is complete and requires little memory. If the best-first search strategy were used, which requires a lot of memory, subgoal reordering would not be necessary.

IV. EXPERIMENTAL RESULTS

A convenient Prolog interface in the prover provides an easy vehicle to carry out subgoal reordering. In the input to the prover, a subgoal of the form prolog(L) presents a call to the Prolog procedure L. We write a Prolog subroutine, called *best subgoal*, to order a list of subgoals according to the evauation function. Another Prolog subroutine is written to translate the standard input format into the format which includes the calls to the Prolog subroutine *best subgoal*. For example, the input clause L:-L_1, L_2, L_3 is translated into the clause

$$L: -\text{prolog}(\text{best-subgoal}([L_1, L_2, L_3], [X_1|Y])), X_1,$$
$$\text{prolog}(\text{best-subgoal}(Y, [X_2, X_3])), X_2, X_3.$$

to perform dynamic reordering; it is translated into the clause

$$L\text{:-prolog}(\text{best-subgoal}([L_1, L_2, L_3], [X_1, X_2, X_3])), X_1, X_2, X_3,$$

to perform static reordering. The resulting clause will be the input to the prover.

We have performed tests on the problem set from [9] using the three heuristics. We tested both static reordering and dynamic reordering using each heuristic on 82 problems. We show part of our experimental results in Tables I–III. We summarize the data in the three tables S1, S2, and S3.[2] As we have expected, no single heuristic, when used for subgoal reordering, performs better on all the test problems. Nevertheless, there are some interesting things revealed by the data.

We first note that subgoal reordering incurs little overhead. This is because the evaluation functions are easy to evaluate, the algorithm for selecting the ordering is simple, and the input clauses in

[2]The data are obtained on a SUN3/60 workstation with 12 megabyte memory. The Prolog system is the ALS Prolog Compiler (Version 0.60) from Applied Logic Systems, Inc.

[3] For example, the two numbers 33 and 15.73 under "improvements for dynamic-H1" indicate that the prover with dynamic subgoal reordering using heuristic H1 does better on 33 of the 82 problems (takes fewer inferences) and the average speedup with respect to the performance of the prover without subgoal reordering is 15.73%; the two numbers 20 and 53.4 under degenerations for dynamic-H1 indicate that the prover with dynamic subgoal reordering using heuristic H1 does worse on 20 of the 82 problems (takes more inferences) and the average slowdown with respect to the performance of the prover without subgoal reordering is 54.4%; the number 29 under *even* for dynamic-H1 indicates that the prover with dynamic subgoal reordering using H1 performs equally well (takes equal number of inferences) as the prover without subgoal reordering on 29 of the 82 problems.

the problems are generally short (7 literals maximal). For the same reasons, dynamic reordering is not more expensive than static reordering. All these can be seen from the data in S1 and S2. The data in S3 suggest that, at least for our heuristics, dynamic reordering should be preferred if subgoal reordering is to be performed at all since dynamic reordering does better on more problems than static reordering using the same heuristics.

The data in S2 suggest that subgoal reordering does not affect the performance of the prover very much. But the data in S1 seem to suggest otherwise. This discrepancy results from the dramatic improvements or degeneration of the performance of the prover when performing subgoal reordering on several problems (ls108, wos15, and wos31). These problems are difficult for the prover without subgoal reordering. This suggests that subgoal reordering can be a valuable addition to the prover for solving hard problems for which we can devise specific heuristics.

One general heuristic does suggest itself. It seems that subgoals with complex structures should be favored. The reasons are exactly those behind H1 and H2. Subgoals with complex structures tend to have small branching factors and can be seen as more important. Special attention should be paid to function symbols since they represent objects in the problem domain. The good performance of the prover when performing subgoal reordering using H2 enforces this rather strongly.

V. Conclusion

It requires domain dependent knowledge to find the optimal ordering for a set of subgoals. In case such knowledge is not available, we have to resort to general heuristics. We have tested several such heuristics and shown that they can have great impact, sometimes adverse, on the performance of a prover. But some heuristics seem to work better or equally well most of the time. Such heuristics are useful since they can make the theorem prover more automatic. We also point out that our heuristics are almost purely syntactic in nature. Heuristics of this sort are simple in form and impose low overhead in their evaluations; and they often provide performance improvements. In general, we think that the importance of the syntactic aspect of mechanical theorem proving is not to be ignored, although it may not play a decisive role in the success of this field in the future.

Acknowledgment

We thank two anonymous referees for their comments on an early version of this paper. We also thank M. Stickel for sending his test problems.

References

[1] C. Chang and R. Lee, *Symbolic Logic and Mechanical Theorem Proving*. New York: Academic, 1973.
[2] H. Gelernter, "Realization of a geometry theorem-proving machine," in *Proc. ICIP*, Paris UNESCO House, 1959, pp. 273–282.
[3] M. R. Genesereth and N. J. Nilsson, *Logical Foundation of Artificial Intelligence*. Los Angeles, CA: Morgan Kaufmann, 1987.
[4] R. E. Korf, "Depth-first iterative deepening: An optimal admissible tree search," *Artif. Intell.*, vol. 27, pp. 97–109, 1985.
[5] R. Kowalski and D. Kuehner, "Linear resolution with selection function," *Artif. Intell.*, vol. 2, pp. 227–260, 1971.
[6] D. A. Plaisted, "Non-Horn clause logic programming without contrapositives," *J. Automat. Reasoning*, vol. 4, no. 3, Sept. 1988.
[7] R. Reiter, "A semantically guided deductive system for automatic theorem proving," *IEEE Trans. Comput.*, vol. C-25, no. 4, pp. 328–334, Apr. 1976.
[8] D. E. Smith and M. R. Genesereth, "Ordering conjunctive queries," *Artif. Intell.*, vol. 26, pp. 171–215, 1985.
[9] M. E. Stickel, "A PROLOG technology theorem prover: Implementation by an extended PROLOG compiler," in *Proc. IJCAI*, Oxford, England, July 1986, pp. 573–587.
[10] T. C. Wang and W. W. Bledsoe, "Hierarchical deduction," *J. Automat. Reasoning*, vol. 3, no. 1, 1987.

A Parallel Algorithm for Solving Sparse Triangular Systems

CHIN-WEN HO AND R. C. T. LEE

Abstract—**In this paper, we propose a fast parallel algorithm, which is generalized from the parallel algorithms for solving banded linear systems, to solve sparse triangular systems. We transform the original problem into a directed graph. The solving procedure then consists of eliminating edges in this graph. The worst case time-complexity of this parallel algorithm is $O(\log^2 n)$ where n is the size of the coefficient matrix. When the coefficient matrix is a triangular banded matrix with bandwidth m, then the time-complexity of our algorithm is $O(\log(m)\cdot\log(n))$.**

Index Terms—**CREW PRAM, cyclic reduction, directed graph model, parallel computation, presubstitution, recursive doubling, sparse triangular linear systems.**

I. Introduction

Since many engineering and natural science problems can be formulated as the problem of computing the solution of a linear system of equations $Ax = b$, efficient algorithms to solve linear systems have always been interesting to a large number of researchers. In recent years, because of the availability of multiprocessor systems as well as vector computers, an avalanche of papers on parallel algorithms for linear systems have been published [1]–[3], [7], [9]–[17], [19]–[22].

In this paper, we shall consider the linear system problem whose coefficient matrix A is sparse and triangular. We propose a parallel algorithm whose worst case performance is $O(\log^2 n)$, where n is the size of matrix A. Thus, our algorithm is superior to that proposed in [21].

This paper is organized as follows. Section I gives an introduction of the problem. Section II introduces the parallel algorithm proposed by Wing and Huang [21]. Our algorithm and its performance analysis are given in Section III. Section IV gives concluding remarks.

II. Previous Results

A triangular matrix $A = [a_{i,j}]$, $1 \le i, j \le n$ is a matrix whose nonzero elements occur only in the lower or upper triangle of the matrix. A sparse triangular matrix is a triangular matrix where there are a few nonzero entries. Without losing generality, we may assume that nonzero elements occur only in the lower triangle and diagonal elements are all 1. (If $a_{ii} \neq 1$ for some i, then we may divide the ith row and b_i by a_{ii} without changing the solution values.)

Let us consider the following sparse triangular system:

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & -2 & 1 & & \\ -1 & & -3 & 1 & \\ -2 & & 2 & & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 1 \end{bmatrix} \quad \begin{aligned} x_1 &= 1 \\ x_2 &= 2 \\ x_3 &= 2 + 2x_2 \\ x_4 &= 1 + x_1 + 3x_3 \\ x_5 &= 1 + 2x_1 - 2x_3 . \end{aligned}$$

To solve this system, Wing and Huang [21] proposed that we may construct a directed graph as Fig. 1. In the diagram, each block

Manuscript received September 9, 1987; revised April 19, 1988. This work was suported in part by the National Science Council under Grant NSC77-0408-E007-04.

C.-W. Ho is with the Institute of Computer and Decision Sciences, National Tsing Hua University, Hsinchu, Taiwan, Republic of China.

R. C. T. Lee is with the National Tsing Hua University, Hsinchu, Taiwan and Academia Sinica, Taipei, Taiwan, Republic of China.

IEEE Log Number 9034551.