An Improved Min-Cut Algorithm For Partitioning VLSI Networks

BALAKRISHNAN KRISHNAMURTHY

Abstract — **Recently, a fast (linear) heuristic for improving min-cut partitions of VLSI networks was suggested by Fiduccia and Mattheyses [6]. In this paper we generalize their ideas and suggest a class of increasingly sophisticated heuristics. We then show, by exploiting the data structures originally suggested by them, that the computational complexity of any specific heuristic in the suggested class remains linear in the size of the network.**

In addition, we present a variety of other techniques that improve the optimality of the partitions produced. Empirical results are provided for various combinations of the suggested improvement schemes. A detailed complexity analysis indicating the run time costs that can be expected for each of the options is also included.

Index Terms — **Partitioning algorithms, VLSI layout.**

I. Introduction

GIVEN a VLSI network, the problem of partitioning the components of the network into two parts of specified sizes so as to minimize the number of signal nets that interconnect components in both parts is called the *network partitioning problem*. This problem arises in various aspects of design automation. For example, it has direct applications in the placement of components during layout (e.g., [3]–[5]). Further, the problem is sufficiently general and admits an analysis independent of its application.

Recently, an extremely fast (almost linear) algorithm was suggested in [6] for this problem. The technique is a modification of ideas in [8] and [9] of improving an existing partition by swapping components between partitions. The selection of the best pair of components to swap often involved searching through a space of c^2 items (where c is the number of components) and such a search might have to be performed c times! Instead, Fiduccia and Mattheyses [6] suggested two modifications: 1) to move only one component at a time, and 2) to keep a sorted list of candidates for moving to the other side. (The idea of moving one component at a time has also been investigated in [10].) They then developed elegant data structures through which they could maintain the sorted candidates, and thus avoid searching for a candidate to be moved.

Their technique has proven to be strikingly efficient; but the limited heuristic of choosing one component, which when moved to the other side will yield maximal improvement in "cut set" size, seems to be somewhat erratic. The quality of the partitions produced seems to be highly influenced by some of the random choices made within the algorithm, such as the order of the components investigated.

Let us illustrate this by an example. Consider two components A and B, both of which are candidates to be moved to the other side, as shown in Fig. 1. Observe that each of A and B will permit the deletion of one signal net (N_1 and N_2) from the "cut set" if it were moved to the other side. Further, the status of nets N_3 and N_4 will remain unchanged by either move. In the terminology of [6], cells A and B are both said to have a gain of $+1$. Since their gains are equal, either one could be selected for the move, arbitrarily. However, B is probably a preferable choice since it would then permit the deletion of yet another net (N_4) through a subsequent move, whereas a similar heuristic in the case of A is less valid.

In this paper we present a technique that captures this "foresight." We develop heuristics that not only encode the above argument but, in fact, capture a generalization of that argument. We then use those heuristics to develop an algorithm preserving those features of [6] that contribute to its efficiency. Consequently, the algorithm presented in this paper not only produces partitions with small cut sizes, it also accomplishes this in an efficient manner. In addition, we show how one can incorporate consistency, i.e., make the algorithm less sensitive to the random choices made within the algorithm.

As pointed out in [9], a network should be viewed as a collection of components (which we shall call *cells*) interconnected by signal nets (which we shall simply call *nets*). A net will be viewed as a set of cells that it interconnects. A positive integral weight will be associated with each cell to denote its size. The size of a set of cells will be the sum of the sizes of its constituent cells. A partition of the cells in the network into two sets $\mathcal{A}$ and $\mathcal{B}$ is called a *simple partition* and is denoted by $(\mathcal{A}; \mathcal{B})$. Its *cut set* is defined as the set of nets that interconnect cells in both $\mathcal{A}$ and $\mathcal{B}$. The partitioning problem asks for a simple partition with a *minimal cut set*.

Partitioning algorithms can be divided into two categories: algorithms that construct a partition from a description of the network, called *constructive algorithms;* and algorithms that improve upon an existing partition, called *refinement algorithms*. A practical approach would be a combination of the two. Our discussion will be primarily on an algorithm of the latter category, while we will briefly mention a technique for the former. Hence, our algorithm will provide a method for improving a given partition.

In order to avoid thrashing during the improvement process, we will require that a cell moved to the other side shall be "*locked*" so that it remains there until "freed." Thus, at an

Manuscript received March 4, 1983; revised August 22, 1983.

The author is with the General Electric Research and Development Center, P.O. Box 8, Schenectady, NY 12301.

0018-9340/84/0500-0438$01.00

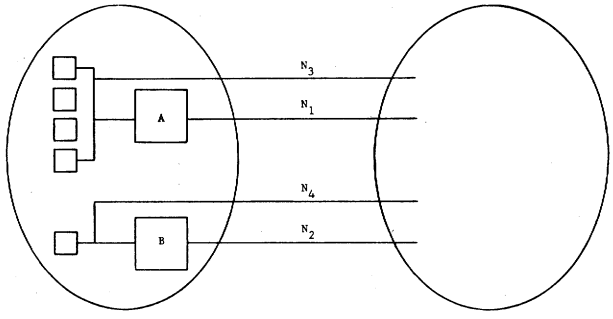

Fig. 1. Competing cells.

intermediate stage of our algorithm, a *partition* (as opposed to a simple partition defined earlier) will be described by a quadruple $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ of sets. $\mathcal{A}_F$ and $\mathcal{A}_L$ are said to belong to the $\mathcal{A}$-*segment* and are, respectively, called the sets of *free cells* and *locked cells*. Similarly, $\mathcal{B}_F$ and $\mathcal{B}_L$ constitute the $\mathcal{B}$-*segment*. Thus, each part of the partition is called a *segment*. An r-partition is a partition in which the ratio of the size of the $\mathcal{A}$-segment to the size of the entire network is r.

For our discussion in this paper, we will consider a network with a set of cells $\mathcal{C}$ and a set of nets $\mathcal{N}$. The number of cells and the number of nets will be represented by c and n. The set of cells on a net $N \in \mathcal{N}$ will be denoted by $\mathcal{C}_N$ and the set of nets incident at a cell C will be represented by $\mathcal{N}_C$. Correspondingly, c_N will denote the size of $\mathcal{C}_N$ and n_C the size of $\mathcal{N}_C$. p will denote the maximal number of nets on any cell, while q will denote the maximal number of cells on any net, i.e.,

$$p = \max_{C \in \mathcal{C}}(n_C)$$

and

$$q = \max_{N \in \mathcal{N}}(c_N).$$

The total number of pins in the network will be denoted by m. Thus, as we shall assume that no net is incident on a cell more than once,

$$m = \sum_{C \in \mathcal{C}} n_C = \sum_{N \in \mathcal{N}} c_N.$$

Finally, χ will denote the cut set size.

We develop the concept of a gain vector in the next section. An overview of the entire improvement algorithm is presented in Section III. In Section IV we present the requirements on the data structures, followed by an account of the results of running our algorithms on real networks in Section V. Section VI is a complexity analysis and in Section VII we conclude with some additional ideas. We provide more detailed algorithms in the Appendix.

II. Definitions

The *incidence number of a net N with respect to a set of cells* $\mathcal{S}$, denoted by $\alpha_{\mathcal{S}}(N)$, is defined as the number of cells in $\mathcal{S}$ that are on net N. That is,

$$\alpha_{\mathcal{S}}(N) = |\{C | C \in \mathcal{S} \quad \text{and} \quad C \in N\}| = |\mathcal{S} \cap N|.$$

Let $\mathcal{A} = (\mathcal{A}_F, \mathcal{A}_L)$ be a segment of a partition. The *binding number of a net N with respect to the segment* $\mathcal{A}$, denoted by $\beta_{\mathcal{A}}(N)$, is defined as

$$\beta_{\mathcal{A}}(N) = \begin{cases} \alpha_{\mathcal{A}_F}(N) & \text{if } \alpha_{\mathcal{A}_L}(N) = 0 \\ \infty & \text{if } \alpha_{\mathcal{A}_L}(N) > 0. \end{cases}$$

A net N is said to be locked to *a segment* $\mathcal{A}$ of a partition if $\beta_{\mathcal{A}}(N) = \infty$, i.e., N is incident on some locked cell in $\mathcal{A}$. N is said to be *locked with respect to a partition* $\mathcal{A}$; $\mathcal{B}$ if it is locked with respect to both $\mathcal{A}$ and $\mathcal{B}$.

Intuitively, the binding number is an indicator of how tightly a net is bound to the segment. A value of ∞ for the binding number indicates that that net must remain incident on that segment. Finally, a net that is locked with respect to a partition must necessarily belong to the cut set of any partition that extends this partition. Before we proceed further, let us make some elementary observations.

Observation 1: If N is a net and $\mathcal{A} = (\mathcal{A}_F, \mathcal{A}_L)$ is a segment of a partition, then[1]

1) $0 \le \beta_{\mathcal{A}}(N)$
2) $\beta_{\mathcal{A}}(N) = \alpha_{\mathcal{A}_F}(N) \iff \alpha_{\mathcal{A}_L}(N) = 0$
3) $\beta_{\mathcal{A}}(N) \le q \iff \alpha_{\mathcal{A}_L}(N) = 0$.

Proof: The first two observations follow from the definitions of α and β. For the third observation, recall that q is the maximum number of cells in any net. Thus, an incidence number with respect to any set of cells can be at most q. The observation then follows. □

We now define similar concepts for the cells in the network. Let $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ be a partition. Consider a cell $C \in \mathcal{A}_F$. Suppose C is on a net N with $\beta_{\mathcal{A}}(N) = 1$ and $\beta_{\mathcal{B}}(N) > 0$. Thus, N appears in the cut set for this partition. If we now move C to the $\mathcal{B}$-segment, we could delete this net from the cut set. On the other hand, if $\beta_{\mathcal{A}}(N) > 0$ (which must be true by the assumptions on C and N) and $\beta_{\mathcal{B}}(N) = 0$, then moving this cell could introduce N into the cut set. In these cases we say that N contributes positively/negatively to the first level gain of C. On a similar note, suppose that $\beta_{\mathcal{A}}(N) = 2$ and $\beta_{\mathcal{B}}(N) > 0$. Then moving the cell C to the $\mathcal{B}$-segment will then permit N to contribute positively to the first level gain of C. On the other hand, if $\beta_{\mathcal{A}}(N) > 0$ and $\beta_{\mathcal{B}}(N) = 1$, then the movement of C to the $\mathcal{B}$-segment will prohibit N from contributing positively to the first level gain of some cell in the $\mathcal{B}$-segment. We then say that N contributes positively/negatively to the second level gain of C.

Motivated by the above argument we define the ith level gain of C, denoted by $\gamma_i(C)$, as

$$\gamma_i(C) = |\{N \in \mathcal{N}_C | \beta_{\mathcal{A}}(N) = i \quad \text{and} \quad \beta_{\mathcal{B}}(N) > 0\}| - |\{N \in \mathcal{N}_C | \beta_{\mathcal{A}}(N) > 0 \quad \text{and} \quad \beta_{\mathcal{B}}(N) = i - 1\}|.$$

Clearly, the ith level gain of any cell C is defined only for $1 \le i \le q$, by Observation 1. In addition, we make the following observation.

Observation 2: For any cell $C \in \mathcal{C}$, $1 \le i \le q$ and any partition of the network

[1]Inequalities involving ∞ will be treated in the usual sense.

$$-p \leq \gamma_i(C) \leq p .$$

Proof: $\gamma_i(C)$ is the difference in the size of two sets, each of which is a subset of $\mathcal{N}_C$. By definition of p, the cell C is on at most p nets. Thus, $|\mathcal{N}_C| \leq p$. The observation then follows. □

As the reader might have guessed, these gain values for the various cells in the network will be compared to determine which cell should be moved to the opposite segment in an attempt to improve the partition. But how many gain values should we compare for every cell? Clearly, the higher level gain values are less important than the lower level gain values. (The algorithm in [6] restricts attention to first level gains.) We will later show that ideally this decision should be determined by the size of the network. Thus, for the present, we shall leave that as a parameter. We then define the *gain vector of order k* of a cell C, denoted by $\Gamma_k(C)$, as

$$\Gamma_k(C) = \langle \gamma_1(C), \gamma_2(C), \cdots, \gamma_k(C) \rangle .$$

Once again, $\Gamma_k(C)$ is defined for $1 \leq k \leq q$.

We now define an order relation "$\leq$" on gain vectors of similar order. $\leq$ is essentially a lexicographic ordering. Thus, $\Gamma_k(C) \leq \Gamma_k(D)$ if either the two gain vectors are identical or the ith component of $\Gamma_k(C)$ is smaller than the ith component of $\Gamma_k(D)$, where i is the index of the first component in which the two vectors differ.

Observation 3: For a fixed k, the order relation "$\leq$" imposes a total order on $\Gamma_k(C)$ for all $C \in \mathcal{C}$.

Proof: Easy. □

Finally, we show a theorem that points out the special role played by the first level gains of cells.

Theorem 1: Let $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ be a partition and C be a cell and χ the current cut set size. If a cell C is moved from one segment to the other, the new cut set size χ' can be expressed as

$$\chi' = \chi - \gamma_1(C) .$$

Proof: Without loss of generality, assume that C is in the $\mathcal{A}$ segment. Let

$$\mathcal{S}_1 = |\{N \in \mathcal{N}_C | \beta_{\mathcal{A}}(N) = 1 \quad \text{and} \quad \beta_{\mathcal{B}}(N) > 0\}|$$

$$\mathcal{S}_2 = |\{N \in \mathcal{N}_C | \beta_{\mathcal{A}}(N) > 0 \quad \text{and} \quad \beta_{\mathcal{B}}(N) = 0\}| .$$

By definition of $\gamma_1(C)$

$$\gamma_1(C) = |\mathcal{S}_1| - |\mathcal{S}_2| .$$

Clearly, every net in $\mathcal{S}_1$ is currently a member of the cut set. Further, by moving C to the $\mathcal{B}$ segment, each of these nets could be deleted from the cut set since C is the only cell on these nets that is currently in the $\mathcal{A}$-segment. Similarly, every net in $\mathcal{S}_2$ is currently not in the cut set and would be added to the cut set if C were moved to the $\mathcal{B}$-segment. Hence, the cut set size, χ', after moving C to the $\mathcal{B}$-segment would be

$$\chi' = \chi - |\mathcal{S}_1| + |\mathcal{S}_2|$$

or

$$\chi' = \chi - \gamma_1(C) . \qquad \square$$

III. Overview of the Algorithm

We are now ready to give an overview of the algorithm. The details of the algorithm are provided in the Appendix and they employ the data structures described in Section IV. Our attempt in this section will be merely to provide an overview of how the algorithm operates. Extensions of this algorithm are suggested and discussed in Section V.

Let us fix k, the order of the gain vectors that we will consider. Given a partition $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ we construct a new partition $(\mathcal{A}'_F, \mathcal{A}'_L; \mathcal{B}'_F, \mathcal{B}'_L)$. To do this we choose a cell C from either $\mathcal{A}_F$ or $\mathcal{B}_F$ with the highest gain vector. To make this choice without searching, we assume that the four sets $\mathcal{A}_F$, $\mathcal{A}_L$, $\mathcal{B}_F$, and $\mathcal{B}_L$ have been maintained sorted according to their gain vectors. (It shall later become clear why the gain of cells in $\mathcal{A}_L$ and $\mathcal{B}_L$ need also be maintained.) Having chosen C, which we shall assume, for the sake of concreteness, is in $\mathcal{A}_F$, we move it to the locked set of the opposite segment, namely, $\mathcal{B}_L$. We now need to insert C into the sorted set $\mathcal{B}_L$. To do this we must compute the new gain vector $\Gamma_k(C)$. (We have tacitly assumed that knowing $\Gamma_k(C)$ would enable us to insert C into the sorted set $\mathcal{B}_L$, without searching for the point of insertion. We justify this in the description of the data structures in Section IV.) To compute $\Gamma_k(C)$ we need $\beta_i(N)$ for $1 \leq i \leq k$ and $N \in \mathcal{N}_C$. For this purpose we maintain an updated table of $\alpha_{\mathcal{S}}(N)$ for $\mathcal{S} = \mathcal{A}_F, \mathcal{A}_L, \mathcal{B}_F, \mathcal{B}_L$, and $N \in \mathcal{N}$. Thus, having moved C from $\mathcal{A}_F$ to $\mathcal{B}_L$, we update the appropriate α values for each $N \in \mathcal{N}_C$. While updating the entries of a net N, we can determine if the gain vectors of other cells on that net are affected by this move. Of course, that decision will depend on the value of k, which we have fixed for the entire algorithm. If the gain vectors of other cells are affected, we adjust the location of those cells in their appropriate sorted lists. In addition, while examining the α values for each $N \in \mathcal{N}_C$, we can also compute the new vector $\Gamma_k(C)$. This will allow us to insert C into $\mathcal{B}_L$ in the appropriate place. That will then complete the moving operation, which we will call MOVE.

Observe that if we now move C from $\mathcal{B}_L$ to $\mathcal{A}_F$ (ignoring the fact that we are moving a locked cell) and perform the suggested updating procedure, we will be back to where we had started! In other words, the operation MOVE is reversible provided we know which cell is to be moved back. This reversibility of MOVE will play an important role in the algorithm.

Continuing the description of the algorithm, we start with a partition of the network in which the locked sets of both segments are empty. We then repeatedly apply MOVE. Each application of MOVE reduces the size of a free set in one of the segments. After each move we can compute the current size of the cut set from the old size of the cut set and the first level gain value of the cell moved using Theorem 1. We continue applying MOVE until both free sets in the two segments are empty. We would then like to retreat to the intermediate partition that had the smallest cut set. This is where the reversibility of MOVE comes into play. However, we need to know the sequence of reverse moves to be made. For this purpose, we should have (and, thus, would have)

kept track of the sequence of cells moved. Having retrieved the best intermediate partition, we free all locked cells and iterate this procedure. Each of these iterations will be called a *pass*.

In the above description we have overlooked the balance requirement between the two segments. When choosing a cell to move, we must require that the move will result in a partition that preserves (approximately) the ratio of the sizes of the two segments. (Note that if such a condition were not imposed, all the cells might migrate to one segment leaving the other empty!) To test this we require that a "balance" condition must be preserved after every move.

This added constraint of having to satisfy the balance condition in choosing a cell to move might force us to search for a cell, instead of automatically taking the cell with the highest gain. However, observe that if one segment is full, the other must be empty. Thus, if we chose the cell with the highest gain from each of the two free sets, one of them must satisfy the balance condition. In fact, even though we do not dictate what the balance condition must be, we do require that the condition has the above property. (This is discussed in Section IV.) Thus, as long as both free sets are nonempty, we are guaranteed a move without searching. On the other hand, this also implies that at the end of the pass both the free sets might not be empty. However, for the balance condition that we shall use, at least one of the free sets will be empty.

This raises an interesting question. If, at some intermediate partition of a pass, more nets have been locked with respect to that partition than the size of the optimal cut set obtained thus far, then it would be futile to pursue that pass any further. For, any subsequent partition constructed in that pass would only be an extension of that intermediate partition, and hence locked nets would remain locked, implying that all those nets would remain in all subsequent cut sets. Thus, we can alternatively terminate the pass when more nets have been locked than the best cut set seen up to that point. The tally of the number of locked nets should then be kept and updated whenever a new net gets locked.

IV. Data Structures

The network can be represented in two ways: through a list of cells for each net, or through a list of nets for each cell. We will, in fact, use both representations. It is easily seen that either of the two representations can be generated from the other in $0(p)$ time. We will make the following assumptions about the network:

1) no net is incident on a cell more than once;
2) every net has at least two cells on it.

Instead of providing the gory details of the data structures, we will merely enumerate the type of information that would be inserted into or extracted from the data structures. We do this by listing a set of primitive operations that we should be capable of performing in *constant time,* independent of the size of the network. For example, the following primitives are required for the two network representations mentioned above:

OP_1: Given a cell C, return $\mathcal{N}_C$. (Actually, this operation might only return a pointer to a linked list representing the set $\mathcal{N}_C$.)

OP_2: Given a net N, return $\mathcal{C}_N$.

We maintain a net table that is capable of providing the following primitives.

OP_3: Given a net N and $\mathcal{S} = \mathcal{A}_F$, $\mathcal{A}_L$, $\mathcal{B}_F$, or $\mathcal{B}_L$, examine/modify $\alpha_{\mathcal{S}}(N)$ and $\beta_{\mathcal{S}}(N)$.

OP_4: Given N, determine if N is locked.

OP_5: Examine/modify the number of locked nets.

For each $\mathcal{S} = \mathcal{A}_F$, $\mathcal{A}_L$, $\mathcal{B}_F$, and $\mathcal{B}_L$, we maintain the set of cells that they represent, ordered according to their gain vectors. This, of course, requires that k, the order of the gain vectors, be specified. Even though k will be fixed, the data structure must be flexible enough so that k can be specified by the user. The primitive operations that are necessary are

OP_6: Return the cell in $\mathcal{S}$ with the highest gain; indicate if $\mathcal{S}$ is empty.

OP_7: Given a cell, delete it from $\mathcal{S}$.

OP_8: Given a cell and its gain vector, insert it in $\mathcal{S}$ at the appropriate place.

The design of a suitable data structure that can provide OP_6, OP_7, and OP_8 requires some clarification. One possibility is to maintain a k-dimensional table with coordinates in each dimension ranging from $-p$ to p. Each entry in this table is a pointer to a doubly linked list of cells whose gain corresponds to the corresponding coordinates. In addition we would need to maintain a direct pointer from each cell to its link in that table.

Finally, we need the following primitive.

OP_9: Given a cell C and source and target sets $\mathcal{S}$ and $\mathcal{T}$ from $\mathcal{A}_F$, $\mathcal{A}_L$, $\mathcal{B}_F$ and $\mathcal{B}_L$, determine if moving C from $\mathcal{S}$ to $\mathcal{T}$ will preserve balance; and if so, update necessary data to indicate such a move.

We will not elaborate on the details of the balance condition. In fact, in [6] the authors suggest experimenting with different balance conditions. For our purposes we will merely assume that the balance condition satisfies the following requirement. For any partition $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ satisfying the balance condition and cells $A \in \mathcal{A}_F$ and $B \in \mathcal{B}_F$, the balance condition must remain satisfied either under the movement of A to the $\mathcal{B}$-segment or under the movement of B to the $\mathcal{A}$-segment.

V. Empirical Analysis and Extensions

In this section we investigate the performance of the proposed algorithm in terms of the optimality of the cut set produced. We defer discussion on the algorithm's time complexity to Section VI. We present results obtained from a pilot implementation of this algorithm. These results have guided the development of further improvements to the algorithm which we present in this section.

In the last section we outlined the requirements on the data structures that would be used. As mentioned in Section I, a partition improvement algorithm, such as the one suggested so far, would require a constructive algorithm to start the improvement process. Our first experiment was to use a random initial partition. The details are suggested in Algorithm 2 (in the Appendix). Combining this with the improvement algorithm, we get Algorithm 5.

This algorithm (as well as others that we shall describe in this paper) was run on some real networks. We observe that a single run would not provide sufficient evidence to compare the results; for each of these algorithms, being heuristic in nature involves making certain arbitrary choices, usually in the form of selecting any one element from a set containing more than one element. Thus, we randomized such arbitrary choices and performed a number of runs. We then compared the range of values obtained for the minimal cut set size. For sake of brevity we merely provide descriptive statistics obtained from the results. These are tabulated in Table I. All results shown were obtained by partitioning the same network into two equal segments. The parameters of this network are $n = 346$, $c = 304$, $p = 7$, and $q = 13$.

The first experiment shows the result of 100 runs using Algorithm 5 with $k = 1$. This is essentially the improvement algorithm suggested in [6]. A striking result of this experiment is the large range of the values for χ, the cut set size. This confirms the observation made in Section I about the random choices that have to be made.

We then increased k to 2 and found both an improvement in the mean as well as in the standard deviation. This is shown in the second row. A further increase to $k = 3$ showed further improvement, but less striking. This indicates that there is a point of diminishing return. We discuss this further in Section VII. Observe that the run times did not increase significantly in increasing k. We provide a more sound argument along these lines in Section VI.

We then employed a technique originally suggested by Kernighan and Lin in [8]. Observe that any algorithm that attempts to find the optimal partition through a sequence of one-cell moves is unlikely to find the optimal partition if it involves the scrambling of a large number of cells. This implies that it might be worthwhile to seek another optimal partition by starting from an "orthogonal" starting point.

More precisely, starting from an initial simple partition $(\mathcal{A}; \mathcal{B})$ of the network we obtain as optimal a partition as we can, say $(\mathcal{A}'; \mathcal{B}')$, through Algorithm 5. We then divide the segment $\mathcal{A}'$ into two parts, $\mathcal{A}_1'$ and $\mathcal{A}_2'$. Similarly, we divide $\mathcal{B}'$ into $\mathcal{B}_1'$ and $\mathcal{B}_2'$. We then explore whether a better partition of the network can be obtained through Algorithm 5 by starting from either of the initial simple partitions $(\mathcal{A}_1' \cup \mathcal{B}_1'; \mathcal{A}_2' \cup \mathcal{B}_2')$ or $(\mathcal{A}_1' \cup \mathcal{B}_2'; \mathcal{A}_2' \cup \mathcal{B}_1')$. If we do find a better partition, we divide that up into two parts and try again.

How are the segments $\mathcal{A}'$ and $\mathcal{B}'$ divided into two parts? Clearly, we wish to divide them in such a way that the two parts have very few nets in common. One approach is to continue recursively. This, however, would render the algorithm inefficient. Instead we use Algorithm 5 (without this improvement) to divide each of the two segments. We refer to this as *KL*-improvement to Algorithm 5. Details of this improved method are given in Algorithm 6.

Results of Algorithm 6 are shown in Table I, with $k = 3$. The improvement, both in the mean as well as the standard deviation, is surprising and encouraging. Observe, however, that the running time increased appreciably. More analytical estimates on the running time are provided in the next section.

TABLE I
DESCRIPTIVE STATISTICS ON THE PERFORMANCE OF THE ALGORITHMS

ALGORITHM'S FEATURES			RESULTS					
INITIAL PARTITION R = RANDOM C = CLUSTERING	k	KL IMPROVEMENT YES/NO	NO. OF RUNS	χ				MEAN RUNNING TIME (CPU SECS)
				MIN	MAX	MEAN	STANDARD DEVIATION	
R	1	NO	100	41	88	67	9.1	3.9
R	2	NO	100	39	72	54	6.9	4.6
R	3	NO	100	39	68	51	6.0	5.2
R	3	YES	20	28	50	40	4.8	47.2
C	1	NO	100	44	68	59	7.2	6.8
C	2	NO	100	36	60	48	5.6	7.8
C	3	NO	100	36	54	44	5.1	8.5
C	3	YES	20	23	44	34	4.5	55.1

Finally, we observe that the starting partition plays a crucial role in the performance of these techniques. Clearly, we wish to find a constructive partitioning technique that would be able to handle those networks on which the improvement technique fares poorly. After some experimentation, we have found that Algorithms 5 and 6 do not fare very well in cases where clustering techniques would have done well. For example, suppose $\mathcal{C}$ and $\mathcal{N}$ are the sets of cells and nets of a network. Let $\mathcal{C}'$ and $\mathcal{N}'$ be identical to $\mathcal{C}$ and $\mathcal{N}$, respectively, except with new names for all cells and nets. Now consider a new network with $\mathcal{C} \cup \mathcal{C}'$ as the set of cells and $\mathcal{N} \cup \mathcal{N}'$ as the set of nets. Clearly, the optimal partition of this hypothetical network into two equal segments would have $\chi = 0$—in particular under the simple partition $(\mathcal{C}; \mathcal{C}')$. However, Algorithms 5 and 6 failed to retrieve this when subjected to this network with the order of the cells scrambled. While this deficiency might not seem crucial since such a network is unlikely to occur in practice, it indicates an inherent weakness in the technique.

To rectify this we suggest using a clustering technique as the initial constructive partition algorithm, in place of Algorithm 2. In Algorithm 7 we provide a simple-minded clustering algorithm. Using this we performed the four experiments again and the results are tabulated in the last four rows of Table I. The results indicate that the clustering technique preprocesses the network so that the arbitrary choices made by the improvement algorithm have a diminished effect on the success of the algorithm.

Summarizing the empirical results, we might note that three factors have played a role in improving the success of our partitioning algorithm.

1) The ability to vary k, the order of the gain vector;
2) The *KL* improvement;
3) Preprocessing the network through a clustering technique.

We have shown that the resulting combined algorithm seems not only highly successful, but consistently successful. In the next section we investigate the time complexity of the algorithm and ask what price one has to pay for each of the refinements suggested in this section.

VI. Complexity Analysis

Clearly, m, the total number of pins in the network, is a measure of the size of the network. In [6] the authors have

shown that their improvement scheme has a time complexity of $O(m)$. Since we claim that for $k = 1$, Algorithm 4 is essentially the same as their improvement technique, we should expect that the time complexity remains the same as well. We show below that for any k, Algorithm 4 has time complexity $O(km)$, which, in the degenerate case of $k = 1$, reduces to the complexity reported in [6].

First, we show that requiring the network to be described through net lists as well as cell lists is not unreasonable.

Lemma 1: Algorithm 1 can be performed in $O(m)$ time.

Proof: If the network is described as a list of nets for each cell, then in one pass ($O(m)$ time) we can produce a cross-referenced list of cells on each net. This will enable us to compute OP_2. OP_1, of course, is available from the original network description itself. □

Lemma 2: Algorithm 2 can be performed in $O(km)$ time.

Proof: Step 1 requires $O(c)$ time. Since $m \geq c$ and $k \geq 1$, it requires at most $O(km)$ time. Step 2 again requires $O(n) \leq O(km)$ time. The statement inside the nested loops of Step 3 requires a fixed amount of time, and the number of times the loop is executed is $\sum_{C \in \mathcal{C}} n_c = m$. Thus, Step 3 requires $O(m)$ time. Likewise, the number of times the loop in Step 4 is executed is $O(m)$. But the statement inside the loop requires $O(k)$ time. Hence, the total time required for the entire algorithm is at most

$$O(c) + O(n) + O(m) + O(k)O(m) \leq O(km)\,. \qquad \square$$

Lemma 3: Except for Steps 3.c and 3.d, Algorithm 3 requires $O(kn_c)$ time.

Proof: Clearly, Step 1 requires constant time and Step 2 requires $O(k)$ time. The loop in Step 3 is executed n_c times. The time required for 3.a, 3.b, and 3.f is constant, while 3.e requires $O(k)$ time. Thus, Step 3 requires $O(kn_c)$ time. Finally, Step 4 requires constant time by assumption of OP_8. In total, Algorithm 3, not counting Steps 3.c and 3.d, requires $O(1) + O(kn_c) + O(1) = O(kn_c)$ time. □

Before we establish the complexity of Algorithm 4, we show a preliminary result.

Lemma 4: If a binding number of a net N ever increases during the execution of Algorithm 4, its value would be ∞ and it will maintain that value for the remainder of the execution of that algorithm.

Proof: Clearly, if a binding number of a net N is ∞, it means that there is a locked cell on that net which will remain locked for the remainder of the algorithm. Thus, that binding number will remain ∞. So, all that remains to show is that whenever a binding number increases, it, in fact, becomes ∞. If that were not the case, then the binding number must have increased through an increase in the incidence number of the net with respect to a set of free cells. But that cannot be since free sets never grow during the course of the algorithm. That completes the proof of the Lemma. □

Theorem 2: Algorithm 4 requires $O(km)$ time.

Proof: First, observe that Step 1 requires $O(1)$ time and Step 3 requires no more time than Step 2 does. Further, even though we cannot estimate the number of times the loop in Step 2 is executed, each time it is executed a cell is moved from a free set in one segment to a locked set in the other segment. Thus, Step 2 can be executed at most as many times as there are cells in the network, which is c. Within Step 2, 2.a and 2.c require only $O(1)$ time.

By Lemma 3, Algorithm 3 requires $O(kn_c)$ time, not counting Steps 3.c and 3.d. Those two steps are invoked only when the following two conditions are met:

1) The net N is incident to the cell moved;
2) Its binding numbers are $\leq k$.

Let us fix a net N and consider how many times the above two conditions would be met during the execution of Algorithm 4. Conceivably, each of the c_N cells on that net could have been moved (but at most once) during the course of the execution. Each time one of those cells is moved, the incidence numbers of N are bound to change, causing a change in the binding numbers. But by Lemma 4, the binding number could only decrease, and if it ever increased, it would become ∞ and remain thus. So, how many times could the binding number of net N have decreased and yet satisfied condition 2? Recalling (from Observation 1) that the binding number is always nonnegative, we conclude that net N could have met the second condition above at most $(k + 1)$ times. Thus, with regard to net N, Steps 3.c and 3.d of Algorithm 3 require $O((k + 1)c_N)$ time. Summing over all nets $N \in \mathcal{N}$, we find that those two steps require in all

$$\begin{aligned}\sum_{N \in \mathcal{N}} O((k+1)c_N) &= O(k+1) \sum_{N \in \mathcal{N}} O(c_N) \\ &= O(k)O(m) \\ &= O(km)\end{aligned}$$

time.

Thus, Step 2 of Algorithm 4 requires

$$\sum_{C \in \mathcal{C}} [O(1) + O(n_c)] + O(km) = O(m) + O(km) = O(km)$$

time. Putting together the time requirements for the three steps in Algorithm 4, we get the following time complexity:

$$O(1) + O(km) + O(km) = O(km)\,. \qquad \square$$

The complexity of Algorithm 5 depends on the number of passes, i.e., the number of calls to Algorithm 4. Our experience has shown that it typically takes three or four passes to converge. (This was also reported by the authors of [6].) If that were so, the entire partitioning algorithm, Algorithm 5, would run in time proportional to km. But for the present we can only hypothesize that it does so in most practical situations, and such a hypothesis seems to hold under our empirical analysis.

Let us look at the complexity of Algorithm 6. Assume for the moment that Algorithm 5 operates within time bound $am + b$. Then Step 3 takes $am + b$ units of time. Steps 4.a and 4.b each take $a\binom{m}{2} + b$ units of time since $\mathcal{A}$ and $\mathcal{B}$ are only half the size of the network. However, 4.d and 4.e each take $am + b$ units of time since those steps involve the entire network. It is not difficult to show that these are the major steps in the algorithm. Now suppose Step 4 is executed L times. Then the complexity of Algorithm 6 would be

$$(am + b) + L\left[2\left(a\binom{m}{2} + b\right) + 2(am + b)\right]$$
$$= (3L + 1)am + (4L + 1)b \simeq (3L + 1)(am + b).$$

If we accept the empirical evidence that Step 4 is executed only two or three times, then by setting $L = 3$, we see that Algorithm 6 should take about ten times longer than Algorithm 5. This is in agreement with the running times observed in our experiments, shown in Table I.

Finally, we claim that the complexity of the simplistic clustering algorithm, Algorithm 7, is linear. The only step that requires clarification is Step 1 where the sorting can be performed using bucket sort (see [1]) which yields a linear complexity bound.

VII. Conclusion

In this paper we have exploited the techniques of [6] by using efficient data structures to create an efficient partitioning algorithm, while at the same time achieving more optimal partitions by using more sophisticated heuristics to guide the improvement of partitions. Particularly significant is the ability to "look ahead" through the heuristics provided by the higher order gains. We have also combined these techniques with the *KL* improvement which often provides significantly better partitions at a surprisingly predictable cost. We believe that a practical implementation should allow the user to weigh the cost against its benefits. Finally, we have pointed out a weakness of such an algorithm and provided, as a remedy, a clustering technique that, in essence, assures that the algorithm is not superficially checkmated. We believe that a more sophisticated clustering technique such as in [2] would be even more helpful. But such claims must await further investigations.

We have suggested a lexicographic ordering on the gain vectors. One could argue for an alternative ordering scheme. For example, suppose that during the improvement of a partition, we have to choose between two cells, A and B, to be moved to the other segment. Suppose the nets on A and B are as shown in Fig. 2. Moving A would eliminate one net; moving B, however, would not eliminate any net, but would then allow for two nets to be eliminated. Their second-order gain vectors would be: $\Gamma_2(A) = \langle 1, 0\rangle$ and $\Gamma_2(B) = \langle 0, 2\rangle$. Which should we rather move? Our philosophy in this paper has been that "a bird in hand is worth two in the bush." Thus, we have ordered $\Gamma_2(A) > \Gamma_2(B)$. But, clearly, there is a case for the reverse order. Such a case might be even more convincing if there were yet additional nets on A and B to make the movement of A not so obviously appealing. However, it is not clear what should replace our lexicographic ordering. We can only say that this requires considerable experimentation.

As a final comment we stress that k, the order of the gain vectors, should be chosen through the parameters of the network. This would then provide more sophisticated heuristics for larger networks. For our examples we studied the size of the set of cells with the highest vector, for various values of k. It is based on this that we settled on $k = 2$ or 3. But it would be impractical to require such a study for every network. Instead, we provide below a technique for choosing a suitable value for k.

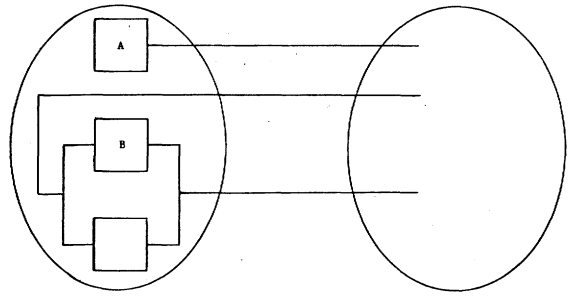

Fig. 2. An example to illustrate alternative orderings on gain vectors.

Recall that p is the largest number of nets incident on any cell in the network. By Observation 2 each component of the gain vector can be any of $2p + 1$ values. Thus, for gain vectors of order k, there are $(2p + 1)^k$ possible vectors. Assuming (even though the assumption might not be precisely true) that at any instant in the algorithm the gain vectors of the c cells in the network are uniformly distributed among the $(2p + 1)^k$ possible values, there would be $c/(2p + 1)^k$ cells with the highest gain. By requiring this number to be 1, we find an approximate value for $k = \log c/\log(2p + 1)$. For our network we find that k should be between 2 and 3!

Appendix

Algorithm 1:
Input: A description of a network.
Output: Two functions, OP_1 and OP_2, as described in Section IV.
Procedure: Trivial.
END

Algorithm 2:
Input: A network described by providing OP_1 and OP_2, a ratio r for the desired partition, and a value for k.
Output: A random simple partition of $\mathcal{C}$ in the ratio r.
Procedure:
1. Assign each $C \in \mathcal{C}$ randomly to either segment $\mathcal{A}$ or segment $\mathcal{B}$ until one of the segments reaches the limit prescribed by the ratio. Assign the remaining cells to the other segment.
2. Initialize the α counts for all the nets.
3. For each cell $C \in \mathcal{C}$ DO
   For each $N \in \mathcal{N}_C$ DO
   Increment the appropriate α counts
4. For each cell C DO
   Examine the α counts of each net $N \in \mathcal{N}_C$ and compute $\Gamma_k(C)$
   Place C in the appropriate cell in $\mathcal{A}_F$ or $\mathcal{B}_F$

END

Algorithm 3:
Input: A partition $(\mathcal{A}_F, \mathcal{A}_L, \mathcal{B}_F, \mathcal{B}_L)$, a cell C to be moved from set $\mathcal{S}$ of segment X to set $\mathcal{S}'$ of segment Y.
Output: A new partition $(\mathcal{A}'_F, \mathcal{A}'_L, \mathcal{B}'_F, \mathcal{B}'_L)$.
Procedure:
1. Delete C from set $\mathcal{S}$.
2. Initialize the gain vector $\Gamma_k(C) = \langle \gamma_1, \gamma_2, \cdots, \gamma_k\rangle$ to all zeros.

3. For each net $N \in \mathcal{N}_C$ DO
   3.a. Adjust the α values of net N
   3.b. Compute $\beta_X(N)$ and $\beta_Y(N)$
   3.c. If $\beta_X(N) \leq k$ then
        For each cell $D \neq C$ in N DO
        Adjust the gain of cell D
   3.d. If $B_Y(N) \leq k$ and $B_X(N) > 0$ then
        For each cell $D \neq C$ in N DO
        Adjust the gain of cell D
   3.e. Add to $\Gamma_k(C)$ the contribution of net N by examining its α values
   3.f. If net N becomes locked by this move, then modify using OP_5
4. The gain vector $\Gamma_k(C)$ has now been computed. Insert C in $\mathcal{S}'$ at the appropriate place.

END

Algorithm 4:

Input: A partition $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ whose cut set size is stored through OP_{10}.

Output: A partition $(\mathcal{A}'_F, \mathcal{A}'_L; \mathcal{B}'_F, \mathcal{B}'_L)$ whose cut set size is stored through OP_{10}.

Procedure:
1. Initialize the sequence of cells moved, σ, to ϕ.
2. While the number of locked nets (obtained through OP_5) $\leq$ best cut set size (obtained through OP_{10}) *and* at least one of the two cells of highest gain in $\mathcal{A}_F$ and $\mathcal{B}_F$ can be moved according to OP_9 DO:
   2.a. Select the cell of highest gain in $\mathcal{A}_F$ or $\mathcal{B}_F$ that can be moved. If both can, then select randomly. Say, A has been selected from $\mathcal{A}_F$. Use OP_9 to indicate the change in balance.
   2.b. Use Algorithm 3 to move A from $\mathcal{A}_F$ to $\mathcal{B}_L$.
   2.c. If the cut set size of the new partition is better than the old cut set size, then $\sigma := \phi$.
        Else $\sigma := \sigma, A$
3. For each cell C in σ (in reverse order) DO
   Move C from its current locked set to the opposite free set

END

Algorithm 5:

Input: A network.

Output: A simple partition.

Procedure:
1. Use Algorithm 1 to prepare the network.
2. Use Algorithm 2 to create a random simple partition $(\mathcal{A}; \mathcal{B})$.
3. Repeat
   Apply Algorithm 4 to $(\mathcal{A}, \phi; \mathcal{B}, \phi)$ and rewrite the result $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ as: $\mathcal{A} := \mathcal{A}_F \cup \mathcal{A}_L, \mathcal{B} := \mathcal{B}_F \cup \mathcal{B}_L$
   Until there is no improvement in cut set size

END

Algorithm 6:

Input: A network.

Output: A simple partition.

Procedure:
1. Use Algorithm 1 to prepare the network.
2. Use Algorithm 2 to create a random simple partition $(\mathcal{A}; \mathcal{B})$.
3. Apply Algorithm 4 to $(\mathcal{A}, \phi; \mathcal{B}, \phi)$ and rewrite the result $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ as: $\mathcal{A} := \mathcal{A}_F \cup \mathcal{A}_L, \mathcal{B} := \mathcal{B}_F \cup \mathcal{B}_L$.
4. Repeat
   4.a. Apply Algorithm 5 to the network restricted to the cells in $\mathcal{A}$ to obtain a simple partition $(\mathcal{A}_1; \mathcal{A}_2)$.
   4.b. Apply Algorithm 5 to the network restricted to the cells in $\mathcal{B}$ to obtain a simple partition $(\mathcal{B}_1; \mathcal{B}_2)$.
   4.c. Set $\mathcal{A}' := \mathcal{A}_1 \cup \mathcal{B}_1, \mathcal{B}' := \mathcal{A}_2 \cup \mathcal{B}_2$, $\mathcal{A}'' := \mathcal{A}_1 \cup \mathcal{B}_2$ and $\mathcal{B}'' := \mathcal{A}_2 \cup \mathcal{B}_1$.
   4.d. Repeat
        Apply Algorithm 4 to $(\mathcal{A}', \phi; \mathcal{B}', \phi)$ and rewrite the result $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ as: $\mathcal{A}' := \mathcal{A}_F \cup \mathcal{A}_L$ and $\mathcal{B}' := \mathcal{B}_F \cup \mathcal{B}_L$
        Until there is no improvement in cut set size
   4.e. Repeat
        Apply Algorithm 4 to $(\mathcal{A}'', \phi; \mathcal{B}'', \phi)$ and rewrite the result $(\mathcal{A}_F, \mathcal{A}_L; \mathcal{B}_F, \mathcal{B}_L)$ as: $\mathcal{A}'' := \mathcal{A}_F \cup \mathcal{A}_L$ and $\mathcal{B}'' := \mathcal{B}_F \cup \mathcal{B}_L$
        Until there is no improvement in cut set size
   4.f. If the result of 4.d. or 4.e. is better than the partition $(\mathcal{A}; \mathcal{B})$, then replace $(\mathcal{A}; \mathcal{B})$ by the result of 4.d. or 4.e.
   Until there is no improvement in cut set size.

END

Algorithm 7:

Input: As in Algorithm 2.

Output: As in Algorithm 2.

Procedure:
1. Sort the nets in decreasing order of their size. Let $\hat{\mathcal{N}}$ be the sorted list.
2. Set $\mathcal{A} := \phi$ and $\mathcal{B} := \phi$.
3. While $|\hat{\mathcal{N}}| \neq 0$ DO
   3.a. Let N be the top net in $\hat{\mathcal{N}}$.
   3.b. Pick either $\mathcal{A}$ or $\mathcal{B}$ whichever is less filled.
   3.c. Assign all cells in $\mathcal{C}_N$ that have not so far been assigned to the set chosen in 3.b.
   3.d. Delete N from $\hat{\mathcal{N}}$,
4. Perform steps 2 thru 4 of Algorithm 2.

END

ACKNOWLEDGMENT

The author is grateful to P. Mellema for the numerous discussions through which many of the ideas reported in this paper have evolved. Thanks are also due to S. Akers for serving as a bouncing board for ideas, C. Fiduccia for a personal account of his algorithm, B. Kwan for much of the programming effort, and F. Wang for supporting this work.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1976.

[2] S. B. Akers, "Clustering techniques for VLSI," in *Proc. IEEE Int. Symp. on Circuits and Systems*, 1982, pp. 472–476.

[3] M. A. Breuer, "Min-cut placement," *J. Design and Fault-Tolerant Computing*, vol. I, no. 4, pp. 343–362, Oct. 1977.

[4] ——, "A class of min-cut placement algorithms," in *Proc. 14th Design Automation Conf.*, 1977, pp. 284–290.

[5] L. I. Corrigan, "A placement capability based on partitioning," in *Proc. 16th Design Automation Conf.*, June 1979, pp. 406–413.

[6] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th Design Automation Conf.*, 1982, pp. 175–181.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability*. San Fransisco, CA: Freeman, 1979, pp. 209–210.

[8] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, pp. 291–307, Feb. 1970.

[9] D. G. Schweikert and B. W. Kernighan, "A proper model for the partitioning of electrical circuits," in *Proc. 9th Design Automation Workshop*, Dallas, TX, June 1979, pp. 57–62.

[10] H. Shiraishi and F. Hirose, "Efficient placement and routing for master-slice LSI," in *Proc. 17th Design Automation Conf.*, Minneapolis, MN, June 1980, pp. 458–464.

Balakrishnan Krishnamurthy received the M.Sc. degree in mathematics from Birla Institute of Technology and Science, India, in 1976, and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts, Amherst, in 1978 and 1981, respectively.

From 1981 to 1983 he was with the Electronics Laboratory, General Electric Company, Syracuse, NY. He is now with the General Electric Research and Development Center, Schenectady, NY. His research interests include combinatorics, complexity theory, theorem proving, and various aspects of design automation for VLSI.

Dr. Krishnamurthy is a member of the IEEE Computer Society and the Association for Computing Machinery.