A Linear-Time Heuristic for Improving Network Partitions

C.M. Fiduccia and R.M. Mattheyses

General Electric
Research and Development Center
Schenectady, NY 12301

Abstract

An iterative mincut heuristic for partitioning networks is presented whose worst case computation time, per pass, grows linearly with the size of the network. In practice, only a very small number of passes are typically needed, leading to a fast approximation algorithm for mincut partitioning. To deal with cells of various sizes, the algorithm progresses by moving one cell at a time between the blocks of the partition while maintaining a desired balance based on the size of the blocks rather than the number of cells per block. Efficient data structures are used to avoid unnecessary searching for the best cell to move and to minimize unnecessary updating of cells affected by each move.

Introduction

Given a network consisting of a set of cells (modules) connected by a set of nets (signals), the mincut partitioning problem consists of finding a partition of the set of cells into two blocks A and B such that the number of nets which have cells in both blocks in minimal. In general, this process is subject to a balancing condition which admits only those partitions whose blocks satisfy a user specified criterion based on size or cardinality constraints.

An exact solution to this problem is currently intractable in the sense that no polynomial-time algorithm for it is known to exist. Since in practice the network may be very large, a practical algorithm must of necessity employ heuristics which exhibit nearly linear running times. This problem has been treated by a number of researchers[1-5] over the last decade. We present an iterative algorithm whose worst case running time, per pass, grows linearly with the size of the network, and which in practice typically converges in several passes. This linear-time behavior is achieved by a process of moving one cell at a time, from one block of the partition to the other, in a attempt to reduce the number of nets which have cells in both blocks. This idea has been independently applied by Shiraishi and Hirose[5]. A technique due to Kernighan and Lin[3] is used to reduce the chance that the minimization process becomes trapped at local minima. Our main contribution consists of an analysis of the effects a cell move has on its neighboring cells and a subsequent efficient implementation of these results.

After specifying the network partitioning problem, we discuss the Kernighan and Lin[3] heuristic and introduce the basic concept of <u>cell gain</u> which is used to select the cell to be moved from one block of the partition to the other. The properties of gain are then exploited to construct a data structure that allows efficient management of changing cell gains. We then address the problem of achieving a desired <u>balance</u> between the sizes of the two blocks of the partition in an environment which allows for differing cell sizes. The problem of determining which cells have their gains affected by each move is then addressed. In both cases, the total amount of work required, per pass, is shown to grow linearly with the size of the network. We close with a discussion of the behavior of a VAX-based implementation of the algorithm by giving the results and the execution times encountered when the program was run on several examples.

The Problem

Following Schweikert and Kernighan[4] we view a network as a set of C cells (modules) cell(1),...,cell(C) connected by a set of N nets (signals) net(1),...,net(N). As far as partitioning is concerned, we may without loss of generality make the assumptions listed below about what comprises a network. We assume that a <u>net</u> is defined as a set consisting of at least two cells, and that each cell is contained in at least in one net. The number of cells in net(i) will be denoted

0420-0098/82/0000/0175$00.75 © 1982 IEEE

by n(i). Any two cells which share at least one net are said to be neighbors. Each cell is assumed to have a size s(i) and a number of pins p(i), indicating that it belongs to exactly that many nets. These assumptions are easily established by the input routine. For input, we assume that the nets are presented one at a time, in any order; each net being completely given before another net is started. Since each pin is on one and only one net, the total number of pins p(1) + ... + p(C), call it P, may be taken as the "length" of the input and, hence, as the "size" of the network. It is clear that neither C nor N will serve this purpose, since neither the number of pins per cell p(i) nor the number of cells per net n(i) is bounded. In any event, both C and N are O(P).

The following input routine will deal with real networks, whose nets are often given as lists of (cell, pin) pairs, which violate some of the above assumptions concerning what constitutes a net. Nets are sequentially numbered 1,2,...N as they are encountered in the input stream. Cells are assumed to be identified by integers in the range 1,2,...C. The principal function performed by the routine is to construct two data structures from the sequence of nets given as input. The first structure is a CELL array, which for each cell contains a linked list of the nets that contain the cell. The second structure is a NET array, which for each net contains a linked list of the cells on the net. In both cases, each linked list created is regarded as a set, with no duplicates and no implicit order. Each record in each of the arrays also contains several additional fields which the algorithm uses to perform its function.

```
/* net-list input routine */
FOR each net n = 1 ... N DO
   FOR each (cell, pin) pair
      (i,j) on net n DO
      /* maintain set property */
      IF net n is not at the front of
         the net-list for cell i
         THEN insert cell i into the
              cell-list of net n and
              insert net n into the
              net-list of cell i
   END FOR
END FOR
```

One should also delete nets with only one cell and a cells that may no longer be on any of the resulting nets. It is clear that O(P) time will suffice to do all of the above work, provided that the number of (cell, pin) pairs in the input stream is O(P).

Given any partition of the cells into two blocks A and B, a net is said to be cut if it has at least one cell in each block and uncut otherwise. Call this the cutstate of the net. This state may be deduced from the net's distribution, this being the number of cells it has in blocks A and B respectively. Define the cutset of the partition to be the set of all nets which are cut. Finally, define the size |X| of a block of cells X to be the sum of the sizes s(i) of its constituent cells.

Given a fraction (ratio) $0<r<1$, we wish to partition the network into two blocks A and B such that $|A|/(|A|+|B|)\cong r$, and such that the size (cardinality) of the resulting cutset is minimized. The ratio r is only intended to capture the balance criterion of the final partition produced by the algorithm. This should not be taken to mean that each move must maintain balance (although this is certainly not ruled out) nor that, in particular, the initial partition need be balanced. We will discuss this point in more detail later. In addition to specifying the ratio r and an initial partition (with one of A or B possibly empty), the user is allowed to designate certain cells as being "fixed" in either block A or block B of the partition. This allows the algorithm to be used to further refine blocks created by previous partitions.

The Basic Idea

Given a partition (A,B) of the cells, the main idea of the algorithm is to move a cell at a time from one block of the partition to the other in an attempt to minimize the cutset of the final partition. The cell to be moved, call it the base cell, is chosen both on the basis of the balance criterion and its effect on the size of the current cutset. Define the gain g(i) of cell(i) as the number of nets by which the cutset would decrease were cell(i) to be moved from its current block to its complimentary block. Note that a cell's gain may be negative. Indeed, g(i) must be an integer in the range -p(i) to +p(i). It is also clear that during each move we must keep in mind the balance criterion to prevent all cells from migrating to one block of the partition. For surely that would be the best partition were balance to be ignored. Thus the balance criterion is used to select the block from which a cell of highest gain is to be moved. It will often be the case that this cell has a non-positive gain. In that case, we still move the cell with the expectation that the move will allow the algorithm to "climb out of local minima". After all moves have been made, the best partition encountered during the pass is taken as the output of the pass. This minimization technique is due to Kernighan and Lin[3].

To prevent the cell-moving process from "thrashing" or going into an infinite loop, each base cell is immediately "locked" in its new block for the remainder of the pass. Thus only "free" cells are actually allowed to make one move during a pass, until either all cells become locked or the balancing criterion prevents further moves. The best partition encountered during the pass is then returned. Additional passes may then be performed until no further improvements are obtained. In practice this typically occurs quickly, in several passes, resulting in a nearly linear algorithm; however, we make no claims about the number of passes required in the worst case, except to point out the obvious fact that, only O(N) passes are possible since the cutset is bounded by the number of nets.

The bulk of the work needed to make a move consists of selecting the base cell, moving it, and then adjusting the gains of its free neighbors. Unless this is carefully done, each cell will have its gain recomputed each time one of its neighbors moves. This is definitely not necessary. The naive approach will lead to an algorithm which performs $(n(i))^2+\ldots+(n(i))^2$ = $O(P^2)$ gain computations per pass. This stems from the fact that the neighborhood relation induced by a net containing n cells is a complete graph with $O(n^2)$ edges. Since a single gain computation for a cell with p(i) pins takes O(p(i)) work, this approach to maintaining cell gains will require more than $O(P^2)$ work. This is particularly expensive even when one large net exists.

We solve the first problem, that of selecting a base cell having the largest gain in its block, by the use of a data structure which quickly returns a cell of highest gain and allows recomputed cell gains to be reentered into the structure in constant time. We consider the solution to this problem in the next section where we discuss the notion of cell gain.

The second problem, that of updating the gains of the neighbors of the base cell, is much more interesting. The naive algorithm consists of recomputing the gain of every free cell on every net of the base cell. We avoid these time consuming pitfalls by showing that a net(i) never accounts for more than 2n(i) gain recomputations during one entire pass. Moreover, we show that each gain recomputation can be replaced by an appropriate sequence of simple gain increment/decrements which can be done in constant time. These solutions to the two problems reduce the total work required to perform one pass to O(P) in the worst case.

Cell Gain

For any partition (A,B) we have defined the gain g(i) of cell(i) as the number of nets by which the cutset would decrease, were cell(i) to be moved from its current block to its complimentary block.

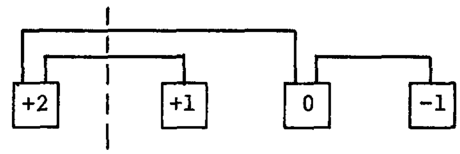

Figure 1. Example of cell gains

Clearly, g(i) is an integer in the range -p(i) to + p(i), so that each cell has its gain in the range -pmax to +pmax, where pmax=max{p(i)|cell(i) is initially free}. In view of the restricted set of values which cell gains may take on, we can use "bucket" sorting to maintain a sorted list of cell gains. This is done using an array BUCKET[-pmax ... pmax], whose k^{th} entry contains a doubly-linked list of free cells with gains currently equal to k. Two such arrays are needed, one for block A and one for block B. Each array is maintained by quickly moving a cell to the appropriate bucket whenever its gain changes due to the movement of one of its neighbors. Direct access to each cell, from a separate field in the CELL array, allows us to yank a cell from its current list and move it to the head of its new bucket list in constant time. Because only free cells are allowed to move, only they need to have their gains updated. Whenever a base cell is moved, it is "locked", removed from its bucket list, and placed on a "FREE CELL LIST" which is later used to reinitialize the BUCKET array for the next pass. This "FREE CELL LIST" saves a great deal of work when a large number of cells have permanent block assignments and are thus not free to move.

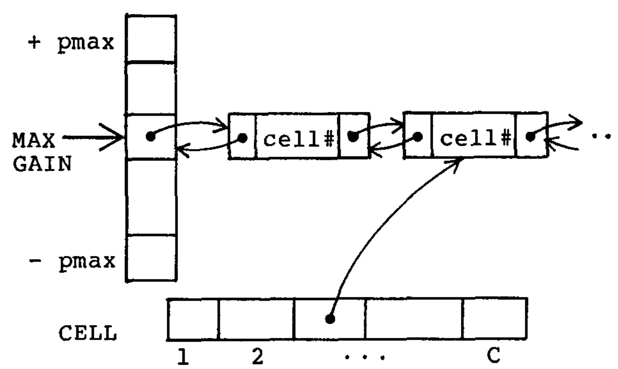

Figure 2. Bucket list structure

For each BUCKET array, a MAXGAIN index is maintained which is used to keep track of the bucket having a cell of highest gain. This index is updated by decrementing it whenever its bucket is found to be empty and resetting it to a higher bucket whenever a cell moves to a bucket above MAXGAIN. Experience with integrated circuit networks shows that gains tend to cluster sharply around the origin and that MAXGAIN moves very little indeed, making the above implementation exceptionally fast and simple. We now establish that, despite its simplicity, this scheme actually does only linear work per pass.

Proposition 1. The total amount of work required to maintain each BUCKET array is O(P) per pass.

Proof. Let f = O(P) be the number of cells in the network which are initially free. Initialization requires O(pmax) + O(f) = O(P) time. If g is the total number of gain adjustments performed during one pass, then O(g) work is sufficient to move all free cells to their appropriate bucket lists, since each cell can be moved in constant time. In the section on maintaining cell gains, we establish that g = O(P). We must finally account for the work required to return a cell of highest gain when one is requested. Let R be the sum of all the amounts by which MAXGAIN is reset by all the various reset actions. Although we cannot in general search and return a cell in constant time, the total time, per pass, used to search down for a non-empty bucket and to return and remove a cell of highest gain is O(R + pmax) + O(f) = O(R) + O(P). In the next section we show that R = O(g); so that O(P) total work, per pass, is sufficient to initialize and maintain the bucket lists. QED

Establishing Balance

The concept of mincut partitioning is meaningless unless a restriction is placed on the sizes of the two blocks; otherwise, we could achieve an empty cutset by moving all of the cells to one block of the partition. The approach we have taken is to specify a fraction (ratio), $0 < r < 1$, to suggest that only final partitions satisfying $|A|/(|A| + |B|) \approx r$ are acceptable. Since in general equality cannot be achieved, some notion of an acceptable tolerance must be incorporated into the balancing scheme. We have considered several approaches, including the use of cost functions based on the size of the cutset and the amount by which the partition deviates from the desired ratio r. We are currently using a scheme which is both fast and seems to work well when the variance in cell sizes is not too large.

Call a partition (A,B) balanced provided that

$$rW - smax \leq |A| \leq rW + smax$$

where $W = |A| + |B|$ is the sum of the s(i), and smax = max{s(i)} is the size of the largest cell which is initially free. A special initial pass is used to establish the balance by moving cells to or from block A depending on the sizes of blocks A and B and the desired ratio r. During this pass, as in all other passes, the base cell is selected according to the highest gain criterion. Once balance is achieved, it is possible to maintain it with every move because the tolerance always allows at least one free cell from either A or B to be moved. If desired, a tolerance of ±k*smax may be used, where $k = k(s) \geq 1$ is some slowly growing function of the number of free cells in the network.

Having established balance, the basic idea of repeatedly choosing a base cell to be moved is described as follows:

1. Consider the first cell (if any) of highest gain from each BUCKET array, rejecting it if moving it would cause imbalance. If neither block has a qualifying cell, no more moves will be attempted.

2. Among those cells returned in step one, choose a cell of highest gain, breaking ties by choosing the one which gives the best balance. Break remaining ties as desired.

3. Return this as the base cell; remove it from its bucket list; and place it on the FREE CELL LIST.

Having chosen a base cell, we now move it to its complimentary block; lock it; and determine the effects it produces on the distributions of its nets and on the gains of its neighboring cells. Unless this is done carefully, the resulting time, per pass, will be worse than $O(P^2)$. We next show how to do this in linear time per pass.

Computating and Maintaining Cell Gains

We have yet to describe how to compute and maintain cell gains. To do this, we must introduce the notion of a critical net. Consider an arbitrary net n. Given a partition (A,B), define the distribution of net n, relative to this partition, as an ordered pair of integers (A(n),B(n)) which represents the number of cells the net n has in blocks A and B respectively. These are clearly computable in O(P) time for all nets. Recalling the definition of the cutstate of a net, we say that a net is critical if there exists a cell on it which if moved would change the net's cutstate. It is easy to see that n is critical iff: either A(n) or B(n) is equal to 0 or 1.

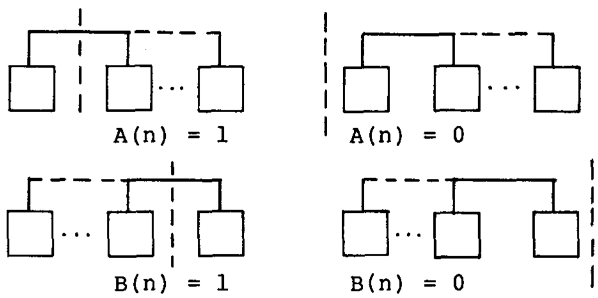

Figure 3. Critical nets

It is now clear that the gain of a cell, previously defined in terms of its effect on the cutset, depends only on its critical nets. This means that if the net is not critical, its cutstate cannot be affected by a move. What is more important, a net which is not critical either before or after a move cannot possibly influence the gains of any of its cells. This observation, coupled with the fact that base cells are "locked" after being moved, will form the basis of our linear-time claim.

Let F ("From") be the current block of cell(i) and T ("To") be its complimentary block; so that F=A and T=B or vice-versa. The gain of cell(i) is then given by

$$g(i) = FS(i) - TE(i),$$

where FS(i) is the number on nets which have cell(i) as their only F cell, and TE(i) is the number of nets which contain cell(i) and have an empty T side. Thus a critical net on cell(i) contributes +1 or -1 to g(i). The following algorithm computes the initial gains of all free cells.

```
/* compute cell gains */
FOR each free cell i DO
   g(i) ← 0
   F    ← the "from block" of cell(i)
   T    ← the "to block"   of cell(i)
   FOR each net n on cell i DO
      IF F(n) = 1 THEN increment g(i)
      IF T(n) = 0 THEN then decrement g(i)
   END FOR
END FOR
```

Proposition 2. Initialization of all cell gains requires O(P) work.

Proof. Making use of the FREE CELL LIST, the outer loop scans through the free cells in the network. For each free cell, the inner loop scans through each of the cell's nets and performs a simple increment or decrement operation. Thus the total work involved is O(rp) = O(P), where rp is the number of pins reachable from all the free cells. QED

Next we prove that a linear amount of time is sufficient to maintain the gains of all free cells during a single pass of the algorithm. Since a net is critical if and only if it contains a cell which if moved would alter the cutstate of the net, we need look at only those nets, connected to the base cell, that are critical before or after the move. Only nets consisting of either two or three cells can be critical both before and after a move. For such nets, two gain adjustment actions might be required: two-cell nets will have one cell incremenetd or decremented twice, whereas three-cell nets will have one cell incremented and another cell decremented.

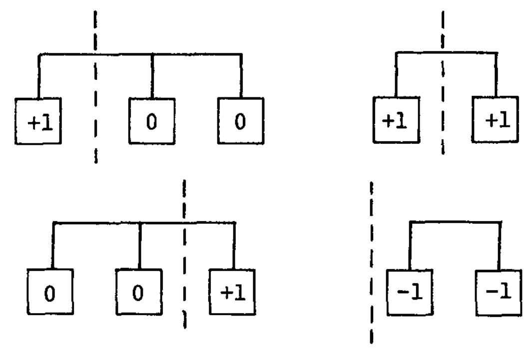

Figure 4. Nets requiring 2 adjustments

If a net is critical, either before or after a move, the contributions it makes to the gains of its cells need to be adjusted. Of course, this should only be done if the net's distribution is changed by the move; that is, only for nets on the base cell. Using the "from-to" terminology of the gain computation algorithm, we see that a net is critical before the move iff

$$F(n) = 1 \text{ or } T(n) = 0 \text{ or } T(n) = 1.$$

The case F(n) = 0 can not occur because the base cell is on the F side before the move. Similarly, a net is critical after a move iff

$$T(n) = 1 \text{ or } F(n) = 0 \text{ or } F(n) = 1.$$

To simplify the situation, we further note that F(n) = 1 before the move iff F(n) = 0 after the move, and that T(n) = 1 after the move iff T(n) = 0 before the move. The following code checks for each of these four cases to see if gain updates are required. A careful analysis of the four cases, which are not independent, will assure the reader that the correct updates are applied.

```
/* move base cell and update
   neighbors' gains */
F ← the "from block" of base cell
T ← the "to block"  of base cell
Lock the base cell and
Compliment its block
FOR each net n on the base cell DO
   /* check critical nets
      before the move */
   IF T(n) = 0 THEN increment gains of
                    all free cells on
                    net(n)
   ELSE IF T(n) = 1 THEN decrement gain
                    of the only T cell on
                    net(n), if it is free
   /* change the net distribution
      to reflect the move */
   decrement F(n)
   increment T(n)
   /* check critical nets
      after the move */
   IF F(n) = 0 THEN decrement gains of
                    all free cells on
                    net(n)
   ELSE IF F(n) = 1 THEN increment gain
                    of the only F cell on
                    net(n), if it is free
END FOR
```

The action of incrementing or decrementing the gains of a specific subset of the cells, on a net consisting of n cells, requires at most O(n) work because, in one scan of the net, each cell can be reached from the net's cell list and can be moved from one bucket to another in constant time. We shall refer to one scan of a net's cell list as an update operation.

Proposition 3. No more than four update operations per net are performed during one pass of the algorithm.

Proof. We first transform the inner loop of the gain update algorithm to simplify the discussion. To do this, we need to distinguish between the free and locked cells of net(n) in each block of the partition. Let LF(n) and FF(n) respectively refer to the number of locked and free cells net(n) has on the F side of the partition. A similar notation is used for the T side. Concentrating on the first conditional in the loop body, notice that T(n) = 0 requires that LT(n) = FT(n) = 0. The condition T(n) = 1 requires that either LT(n) = 1 and FT(n) = 0, or that LT(n) = 0 and FT(n) = 1; however, the update is performed only if the cell on the T side is free; that is, only if LT(n) = 0. Using this observation, and a similar observation for the conditional updates after the distribution shift, the code for the inner loop of the gain adjustment algorithm can be restated as:

```
/* check for critical nets
   before the move */
IF LT(n) = 0
   THEN IF FT(n) = 0 THEN "update gains"
   ELSE IF FT(n) = 1 THEN "update gains"
/* change the net distribution
   to reflect the  move */
decrement FF(n)
increment LT(n)
/* check for critical nets
   after the move */
IF LF(n) = 0
   THEN IF FF(n) = 0 THEN "update gains"
   ELSE IF FF(n) = 1 THEN "update gains"
```

Observe that once both blocks A and B have served in the capacity of the T side for a given net n, no further update operations will occur for that net. This is because the code which updates the net's distribution will have incremented the locked cell count on both sides. Once this occurs, the net is essentially "dead", meaning that its cutstate can no longer change, thus ruling out the possibility of future updates.

This observation allows us to concentrate on only that portion of the move sequence, for an individual net n, which includes the first change in direction of cell movement. We will consider a sequence of moves (with respect to the net n) of cells from the A side (A-move) followed by a single move of a cell from the B side. During the first A-move T=B, thus for all subsequent moves LB(n) will be positive. Therefore, the B side, having only 0 or 1 cells, can only cause an update on the first A-move of the sequence. During the sequence of A-moves, each move causes the FA(n) component of the net distribution to be decremented by one. Updates can occur only for values of FA(n) = 1 and FA(n) = 0, and only once for each value with F=A. The final move with B=F could also cause an update if the A side has 1 or 0 cells. Since no further updates can be required, we get a total of at most four updates per net. A more careful analysis reveals that three updates will be sufficient for any net, and that three updates are necessary for certain nets. During these three updates, the gain g(i) of a given cell(i) is adjusted at most twice. QED

Using facts from the previous proof, we can now complete the proof of Proposition 1. We see that g, the total number of gain adjustments per pass is O(f), where f is the number of initially free cells. Thus g = O(f) = O(P) in Proposition 1. Each time a net is updated, the gain of any cell on that net can be incremented at most twice, by Proposition 3; thus, during one update, the value of MAXGAIN can be reset to at most MAXGAIN + 2.

This shows that R, in Proposition 1, is O(N) = O(P). This establishing that the bucket lists can be maintained with O(P) work per pass. QED

We are now in a position to establish the behavior of the our algorithm for maintaining cell gains.

Proposition 4. The total work required to initialize and maintain cell gains is O(P) per pass.

Proof. The total amount of work required for gain maintenance during one pass of the algorithm is the sum of the work required for each individual net. Each update of net(i) uses O(n(i)) work. Proposition 3 shows that only a constant number of updates are required, per net per pass; Since n(1) +...+n(N) = O(P), the linear behavior is obtained. QED

Combining Propositions 1 and 4, we may now state our main result.

Theorem. The minimization algorithm requires O(P) time to complete one pass.

Performance and Applications

The algorithm has been implemented in the language C, and runs on a VAX 11/780. Its performance was evaluated by using it to partition several random-logic polycell designs. Four samples are listed below. The average chip has 267 cells, 245 nets, and 2650 pins. On these chips, the algorithm typically makes about 900 moves per cpu-second. This will of course depend on the average number of pins per cell and the sizes of the nets. The factor by which the algorithm will outperform the naive algorithm depends on network size and especially on the size of the largest nets. The new algorithm is superior especially when the network contains even one large net.

	CELLS	NETS	PINS	PASSES	TIME
Chip 1	306	300	857	3	1.63
Chip 2	296	238	672	2	.98
Chip 3	214	222	550	5	1.91
Chip 4	255	221	571	5	2.09

As a cell placement tool, in a polycell environment, the algorithm is being evaluated in two quite distinct ways. The first is a straight-forward application to partition the cells into channels. We call this inter-channel placement. Its objective is to reduce the number of inter-channel connections needed. The second application is as an intra-channel placement tool. Here the objective is to reduce channel density and wire length. This is done recursively to determine first, in which half of the channel the cell should be placed, then in which quarter, and so on. We feel that this is a novel approach to intra channel placement.

Acknowledgements

The authors wish to thank Bob Darrow, who implemented the algorithms on the VAX. Without the feedback one gets from such implementations, it is difficult to evaluate a heuristic solution. Thanks are also due to Phil Lewis and Ron Rivest for their suggestions.

References

[1] M.A. Breuer, "Min-Cut Placement," J. of Design and Fault- Tolerant Computing, Vol.I, number 4, Oct. 1977, pp. 343-362.

[2] M.A. Breuer, "A Class of Min-Cut Placement Algorithms," Proc. 14th Design Automation Conference, New Orleans, 1977, pp. 284-290.

[3] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," Bell System Technical Journal, Vol. 49, Feb. 1970, pp. 291-307.

[4] D.G. Schweikert and B.W. Kernighan, "A Proper Model for the Partitioning of Electrical Circuits," Proc. 9th Design Automation Workshop, Dallas, June 1979, pp. 57-62.

[5] H. Shiraishi and F. Hirose, "Efficient Placement and Routing for Masterslice LSI," Proc. 17th Design Automation Conference, Minneapolis, June 1980, pp. 458-464.