The Min-Cut Shuffle: Toward a Solution for the Global Effect Problem of Min-Cut Placement

Inderpal Bhandari, Mark Hirsch and Daniel Siewiorek

Department of Electrical and Computer Engineering
Carnegie Mellon University

Abstract

The Global Effect Problem is introduced in the context of adapting Min-cut partitioning for Min-cut placement. A simplified version of the problem is solved in linear time by using a new algorithm, the Min-cut Shuffle. A detailed analysis and implementation for the algorithm is presented.

1. Introduction

Circuit layout refers to the combination of placement and routing. Although partitioning is frequently mentioned as another task that is fundamental to layout, it is merely a problem reduction technique. Because circuit layout belongs to the NP-hard class of problems, a solution that does not apply a problem reduction technique is likely to prove disastrous in terms of run time and quality of layout. Top-down partitioning algorithms recursively slice a design into parts until only primitives remain, thus dividing the scope of the layout problem in two ways:

- The number of interdependencies between derived subproblems are identified and minimized. This will be referred to as the Interdependency Minimization Effect (IME).
- Relative positions for clusters are proposed to such extent that they be physically contained within their parent cluster. This will be referred to as the Logical Placement Effect (LPE).

The class of algorithms which considers the wires crossing partition boundaries as an interdependency is known as Min-cut. The scope of this paper is limited to the adaptation of min-cut partitioning (MP) to do placement.

The rest of this paper is organized as follows. Section 2 describes fundamental problems with Min-cut placement and outlines previous approaches with their limitations. Section 3 defines the problem statement addressed by this work. An algorithm and its implementation that confronts the problem statement is described in Section 4.2. Section 5 discusses future directions and summarizes the contributions of this work.

2. Fundamental Problems in Min-cut Placement

Placement, unlike partitioning, must model the physical carrier. For the purpose of this work, the carrier will be viewed as a grid, a model that has been widely used for describing algorithmic work in layout [1, 6]. The individual places of the grid will be referred to as slots. Placement is, therefore, the assignment of circuit primitives to slots. Min-cut placement (MPLA) uses MP in a top-down manner. An example is given below to demonstrate the technique and illustrate IME and LPE.

Min-cut partitioning is accomplished by placing circuit primitives on either side of an imaginary slice through a cluster in such a manner as to minimize the number of wires crossing the cut. The process is recursively repeated for each of the two smaller clusters until only primitives remain. Several issues affect the quality of placements produced by MP techniques: First, MP is itself in the NP-hard class of problems. Powerful heuristics are necessary for it to be used. Secondly, MP specifies the partitioning of clusters but provides limited guidance as to where the clusters should be placed. LPE imposes placement restrictions only on parent-child cluster pairs and even in these cases actual assignment to slots is not done. A technique is needed to control the placement of clusters. Finally, IME eliminates dependencies between sub-problems if placement is considered to be logical recursive partitioning. However, this is not true since assignment of clusters to slots must be done. It turns out that if assignment to slots of clusters is interleaved with partitioning, then IME minimizes the number of cut lines going across partitions but in general does not address global interdependencies such as total wirelength as illustrated in Figure 2-1.

A technique is needed to place clusters produced by MP, to decide the ordering of partitioning for the children and to use the interdependency information between subproblems to advantage. This will be referred to as the Global Effect Problem.

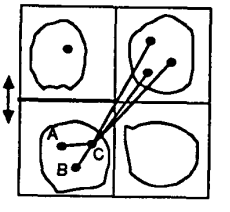


Figure 2-1: Global Dependencies

Fiduccia and Mattheyses [3] introduced a powerful technique (henceforth referred to as FM) to partition electrical circuits. The major achievement of FM was its linear time complexity. [4] showed that FM was actually a special case of a class of algorithms all of which shared the same time complexity and displayed increasing sophistication.

Two techniques have been used to combat the Global Effect Problem. [2] introduces the notion of dummy nodes, a clever device to deal with global constraints, and a similar idea is expressed as the concept of 'inplace partitioning' presented in [5]. Both ignore the order of partitioning, a weakness that can result in bad partitions as illustrated in Figure 2-2.

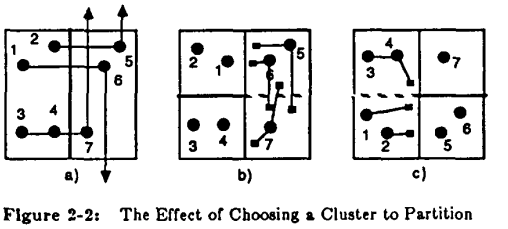


Figure 2-2: The Effect of Choosing a Cluster to Partition

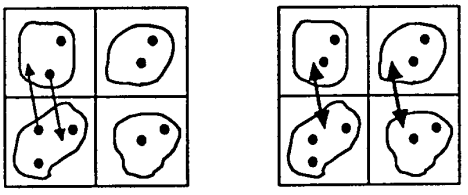
Figure 2-2a shows a partitioning with the local and global

interdependencies. The circles represent circuit elements, the lines represent interconnecting wires and the arrows point to circuit elements in other partitions. Figure 2-2b shows the partitioning that would result if the left partition again was partitioned before the right one. The squares are introduced to represent dummy nodes. Figure 2-2c shows the partitioning that would result if the order of partitioning was reversed from that in Figure 2-2b. Figure 2-2 demonstrates that the order in which clusters are partitioned clearly affects the resulting partitions and is a decision point that is arbitrarily decided in existing approaches.

3. Problem Statement

Assume that all partitioning is done using FM, and the clusters returned at every level are to be placed simultaneously using global optimization. Devise a procedure to do this in linear time.

Note that the above problem statement is a simplified version of the Global Effect problem in that it does not deal with local and global constraints simultaneously. However, it does use interdependency information between subproblems advantageously so that the placement of clusters in MPLA is not an arbitrary decision. Furthermore, the technique for solving the problem statement may apply directly to the Global Effect Problem. The problem statement is illustrated in Figure 3-1. Figure 3-1a shows the effects of an application of FM. FM deals with local constraints while forming the partitions. The partitions are formed by moving circuit elements across the cut line as indicated by the arrow. The global constraints are dealt with when the partitions are positioned about the cut line as indicated in Figure 3-1b. The simplified problem statement will be referred to as the Min-Cut Shuffle (MCS), since it optimizes a global metric by *shuffling* the partitions back and forth across their mutual cut lines.



a) b)

Figure 3-1: Simplified Global Effect Problem

4. Algorithm

Some definitions are necessary to describe the algorithm:

- Slicing tree--The partitioning sequence may be represented as a binary tree, B. Furthermore, each tree node may be associated with a supercell specifying the circuit elements in the partition and a superslot specifying on which physical block the partition resides on the carrier. Figure 4-1a shows a slicing tree that has been sliced vertically into a left and right side, and each side has been sliced horizontally. The partition index (PI) specifies the level of the tree that is currently being partitioned.

- Slot--A legal placement site. Slots K and L are *neighbors* iff they share a common boundary.

- Supercell--An electrical circuit will consist of primitives and nets. Clusters of primitives will be referred to as supercells. Figure 4-1b shows an example of a supercell, N_d. Two supercells are *siblings* if they have the same parent supercell. A *move* is the swapping of two siblings across their mutual cut line.

- Superslot--A set of slots, P, is a superslot iff there exists a path for every pair of slots in C and all slots in such a path are also in P. A string of slots $s_1, s_2, \ldots, s_n$ is a path for the pair s_1, s_n iff s_i and s_{i+1} are neighbors for all such i. Note that n may be one. An example of a superslot is labeled *LT* in Figure 4-1c. Superslots are vertically or horizontally *adjacent* to each other if they share a horizontal or vertical cut line respectively.

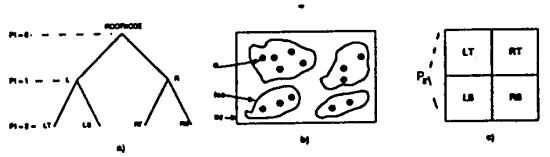


Figure 4-1: Definitions

4.1. Data Structures

The shuffle algorithm uses data structures similar to [3]. The network at any tree level can be represented as a list of supercells connected to each net and as a list of nets connected to each supercell. A net is connected to a supercell iff it is connected to primitives in two or more supercells. In addition, the algorithm requires a data structure for representing the gains of the various moves in order to facilitate identifying the move with the highest gain and appropriately update the gains of remaining moves. A bucket structure was proposed by Fiduccia and Matthyeses [3]. It is an array BUCKET[-pmax .. pmax] whose kth entry contains a doubly-linked list of moves with gains currently equal to k. Pmax is the total number of pins in the circuit, a pin being defined as the connecting point of a primitive to a net. The relationships between the list of supercells, list of nets, and bucket array is illustrated in Figure 4-2.

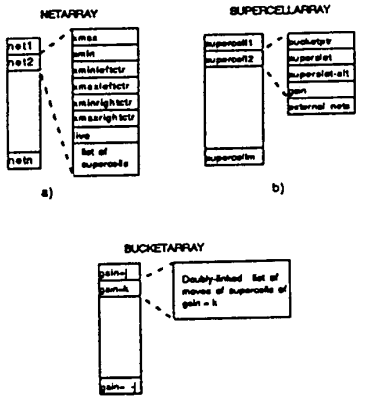


Figure 4-2: Data Structures

The total estimated wirelength was chosen as the cost metric for a given placement configuration. The wirelength is estimated as one half the perimeter of a box bounding the centers of all supercells connected to the net. For any PI that is shuffled, all moves will be in the same direction, either vertical or horizontal. The algorithm will be described only for moves in the horizontal direction without loss of generality. When considering horizontal moves, the estimated wirelength is given in terms of distance units between the midpoints of supercells at the horizontal endpoints of the net. The wirelength estimate changes by an amount proportional to the distance between the centers of the two adjacent supercells involved in the move. Hence the *gain* of a move is simple to record; it is the number of nets that are shortened/lengthened (assuming all supercells are the same size). It is now easy to see that the gains will always be in the range -pmax to pmax.

The complexity of the gain update calculation is reduced by recording the extremities of the bounding box for a net along with the co-ordinates of the horizontal adjacent superslots that swap with these extremities and a count of the number of supercells which lie at these extremities. Figure 4-3 shows the information which describes a net and a supercell, and Figure 4-4 illustrates it. In the former figure, NET and SUPERCELL are the elements in arrays NETARRAY and SUPERCELLARRAY respectively. Therefore, for any supercell or net, it is possible to index its SUPERCELL or NET datastructure.

```
NET {
list_of_supercells--supercells connected to NET
xmin--smallest x-coordinate for NET
xmax--largest x-coordinate for NET
xminleftctr--number of supercells at left child tree
   nodes and located at xmin(NET)
xminrightctr--number of supercells at right child
   tree nodes located at xmin(NET) or may be
   swapped to xmin(NET).
xmaxleftctr--number of supercells at left child tree
   nodes located at xmax(NET) or may be swapped
   to xmax(NET).
xmaxrightctr--number of supercells that are at right child
   tree nodes and are located at xmax(NET).
}

SUPERCELL {
list_of_nets--list of nets connected to SUPERCELL
xcoordinate--the x coordinate of SUPERCELL
}
```

Figure 4-3: Net and Supercell: Partial Data Structures

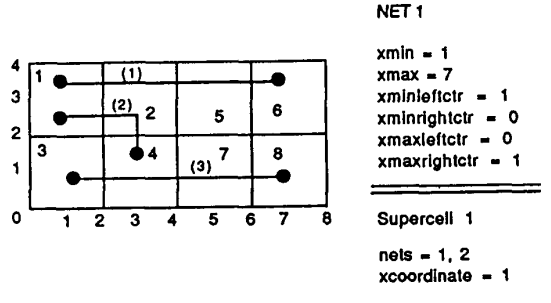


Figure 4-4: Illustrated Net and Supercell: Partial Data Structures

4.2. Algorithm: Detailed Description

An overview of the algorithm is given in Figure 4-5 and consists of two parts, initialization and iteration. The iterative step is referred to as a pass. It consists of updating the gain array bucket structure, and then iteratively choosing the move with the highest gain, removing it from the list of potential moves and updating the affected gains of moves remaining on the list until there are no remaining moves. A supercell that has been moved in a pass is *locked* for that pass; otherwise it will be *free*. The best placement configuration for the pass becomes the starting position for the next pass. Passes are made until there is no further improvement of gain.

```
Shuffle()
   initialize
   Do Until (no improvement for a pass)
      initialize gain array bucket structure
      save state of the physical positions of the partitions
      Do Until (no moves remain in gain array bucket structure)
         accept the move with the highest gain
         remove the move from the gain array bucket structure
         adjust partition gains affected by the move and update
               gain array bucket structure
         If (gain > 0) then
            save assignments of physical positions of partitions
   restore the physical positions of the partitions
```

Figure 4-5: Overview of the Shuffle Algorithm

Initialization:

1. Record the arbitrary assignment of supercells to superslots as returned by FM. The time complexity is O (*P*) where P denotes the number of primitives in the design.

2. Initializing NETARRAY and SUPERCELLARRAY is bounded by O (*pmax*). This follows from the fact that all information needed can be obtained by accessing in the worst case all supercells connected to all nets. Generating buckets and pointers to buckets in BUCKET is bounded by O (*P*). Before buckets may be properly located, the initial gains of all moves must be calculated with the algorithm given in Figure 4-6. The algorithm is executed for each supercell and returns the gain contribution to a move involving the supercell. The gain of a move is the sum of the gains of the associated supercells. The following notation is used to describe the algorithm:

- ++ and -- indicate that the preceding variable is incremented or decremented by 1 respectively. For instance, x++ increments x.
- Xcoordinate(n)--The x coordinate of a supercell n.
- Gain(n)--Store and update the gain of the move associated with supercell n.
- *Left and right child nodes*--The superslot assignment of sibling supercells. For instance, if the supercell is assigned to a left child superslot, it is referred to as a left child node.
- Live net--A net is live with respect to a supercell n iff it is connected to n but not to its sibling.

It is easy to show that the initial gains of all supercells and the proper location of all buckets can be done in O (*pmax*), as a constant number of operations are involved and no operation takes more than O (*pmax*).

3. Calculation of live nets. The calculation of all live nets for a pair

nets on the right sibling. It can be derived fairly easily from the given algorithm and hence is omitted. Jointly, the algorithms will be referred to as UPDATE. UPDATEL is self-explanatory. Note that except for the cases where all supercells connected to some net have to be updated (henceforth referred to as net-updates) and the computation associated with maintaining BUCKET, it is clear that the complexity of the procedure is bounded by O (*pmax*). Also, the structure of UPDATEL can be split into two parts, associated with changes in the maximum and minimum co-ordinates of a net. These parts will be referred to as MAXUPDATEL and MINUPDATEL respectively. Corresponding parts may be identified in UPDATER. An easy observation that follows directly from the structure of UPDATEL is that the gain of any move can be increased at most by 2 if it is affected by a net-update. This means that MAXGAIN can increase at most by 2 for a net-update. Another easy observation is that for any net-update involving net n the computation is bounded by O ($n(i)$), where n(i) is the number of supercells connected to net n. It is shown below that for any pass all computation associated with maintaining BUCKET is bounded by O (*pmax*) and that at most six net-updates can occur for any net. It is then obvious that the complexity of UPDATE and of MCS is O (*pmax*).

3. Maintaining BUCKET in any pass is bounded by O (*pmax*). BUCKET is used in UPDATE when gains of moves are changed. Inserting and deleting buckets from lists can be done in constant time [3]. Setting of the MAXGAIN pointer so that it points to the list of buckets with the maximum gain. In the case of an increase in the available maximum gain this only involves indexing the BUCKET array, as the new maximum gain is known. In the case of a decrease in the available maximum gain it may happen that the list of buckets at that gain becomes empty. To reset MAXGAIN it is necessary to scan down (henceforth referred to as SCANDOWN) from the old maximum gain to the first gain that has a non-empty list of buckets. Note that maximum gain cannot exceed pmax and that for any move during a pass MAXGAIN may at most be moved up two units (proved above). As the number of moves in a pass is less than the number of supercells, it is easy to show that the complexity of all SCANDOWN's has to be bounded by O (*pmax*).

4. For any net at most six net-updates can occur during a pass. Associate the properties MINOVER and MAXOVER with a net. A net n is MINOVER if xmin(n) = x co-ordinate of q where q is connected to n, q is a left sibling and q is locked. No further moves can affect xmin(n). A net n is MAXOVER if xmax(n) = x co-ordinate of q where q is connected to n, q is a right sibling and q is locked. No further moves can affect xmax(n). This information can be stored in NETARRAY[n] and can be easily updated without affecting the complexity of the algorithm. It will now be proved that for any pass, the net-updates in MINUPDATE may be executed at most three times for any net. A similar proof can be completed for the net-updates in MAXUPDATE (this is omitted). First MINUPDATE has to be enhanced. Note that once n is MINOVER, there is no need to execute the net-updates for n in MINUPDATEL. Therefore the check: if (xminlctr(n_j) EQ 1) etc. should read: if ((xminlctr(n_j) EQ 1) AND (n_j is not MINOVER)) etc. The second net-update in MINUPDATEL needs no change as if n_j is MINOVER then xminlctr(n_j) will not be 0. (In fact it is not necessary at all to execute MINUPDATEL for a net that is MINOVER, but it is good programming practice to keep the counters correctly updated). Similar arguments apply to MINUPDATER and are omitted here. All that remains to be shown is that the number of net-updates in of siblings i,j may be done in O (p_i+p_j), where p_k is the number of nets connected to supercell k. This is shown as follows: The calculation of live nets for all siblings may be done in O (*pmax*) with a data structure similar to NETARRAY. An array, NTARRAY, with an element corresponding to every net is initialized to 0. A simple algorithm may now be used.

a. Scan through the list of nets in SUPERCELLARRAY[i] and place a 1 in NTARRAY[n] for all n that are in this list.

b. Scan through the list of nets in SUPERCELLARRAY[j]. For every net n, if a 1 appears in NTARRAY[n] then place a 2 in NTARRAY[n].

c. Scan through the list of nets of i and j in that order. For every net n, if NTARRAY[n] = 1 then place a LIVE in the element corresponding to n in the list of nets being currently scanned, else place DEAD.

d. Scan through the list of nets of i and j in that order. For every net n, set NTARRAY[n] to 0.

The linear time complexity of the algorithm is attainable since NTARRAY also has been re-initialized and is ready for the next pair of siblings.

```
Initialize node_gains(Nd_i)
   gain(Nd_i) = 0;
   for (all live nets, n_j on Nd_i)
      If (xmax(n_j) EQ xcoordinate(Nd_i)) then
         If (Nd_i is a left child node) then
            gain(Nd_i)--

         else  /* Nd_i is a right child node */
         If (xmaxright_ctr(n_j) EQ 1)  then
            gain(Nd_i)++

      If (xmin(n_j) EQ xcoordinate(Nd_i)) then
         If (Nd_i is a left child node)  then
            If (xminleftctr(n_j) EQ 1)  then
               gain(Nd_i)++

         else  /* Nd_i is a right child node */
               gain(Nd_i)--
```

Figure 4-6: Algorithm to Calculate Initial Node Gains

Making a Pass: The complexity analysis for completing a pass is detailed below.

1. Reinitializing the data structures between passes is simpler than initialization. The complexity is bounded by O (*pmax*) and follows in a straightforward manner from the arguments presented for initialization of the first pass. The number of passes is bounded by the maximum possible improvement which is bounded by pmax. Saving the best placement can be done in O (*pmax*).

2. All that remains to be shown is that the updates of gains is of O (*pmax*). It is no surprise that this proof is similar to its counterpart in [3], since the data structures and the technique used for MCS are similar to those in [3]. The algorithm to update gains after a move is partially given in Figure 4-7. The given algorithm (henceforth referred to as UPDATEL) updates the gains of moves associated with supercells that are connected to nets on the left sibling. A similar algorithm (UPDATER) is necessary to consider

MINUPDATE that may occur for any net n is at most 3 before it is MINOVER. A move that makes a supercell with x co-ordinate equal to xmin(n) and connected to n a left sibling immediately makes n MINOVER. Such a move (henceforth referred to as RIGHT-LEFT) cannot cause any of the net-updates in MINUPDATEL to be executed for n, but it can cause one net-update for n in MINUPDATER. This will occur as all supercells connected to n which could be swapped to make n MINOVER will have the gains of their corresponding moves incremented by 1. At most two net-updates can be executed in MINUPDATE for n before the execution of a RIGHT-LEFT for n. As the move affecting n cannot be a RIGHT-LEFT all net-updates have to be executed in MINUPDATEL. These can only be executed if the xminlctr(n) = 1 or xminlctr(n) = 0. Disallowing the possibility of a RIGHT-LEFT, it is easy to see that these values for xminlctr(n) can only occur once apiece. This completes the proof.

```
XL    x-coordinate of Mdleft
XR    x-coordinate of Mdright

Update gains after a move(Mdleft, Mdright)
    Lock Mdleft:
   for (all live nets, nj on Mdleft) do

/* MAXUPDATEL */
      if (xmax(nj) EQ XL)  then
         xmaxlctr(nj)--
         xmaxrctr(nj)++
         xmax(nj) = XR
      if (xmaxlctr GT 0) then
         for (all free supercells, Mdk on nj) do
            if (xcoordinate(Mdk EQ XL) then
               gain(Mdk)++
      else if (xmax(nj) EQ XR) then
         if (xmaxrctr(nj) EQ 1) then
            for (all free supercells, Mdk on nj) do
               if (xcoordinate(Mdk) = XR) then
                  gain(Mdk)--
         xmaxlctr(nj)--
         xmaxrctr(nj)++

/* MINUPDATEL */
   if (xmin(nj) EQ XL) then
      xminlctr(nj)--
      xminrctr(nj)++
      if (xminlctr(nj) EQ 1) then
         for (all free supercells, Mdk on nj) do
            if (xcoordinate(Mdk) EQ XL)
               gain(Mdk)++
      if (xminlctr(nj) EQ 0) then
         xmin(nj) = XR
      if (xminrctr(nj) GT 1) then
         for (all free supercells, Mdk on nj) do
            if (xcoordinate(Mdk) EQ XR) then
               gain(Mdj)--
```

Figure 4-7: Algorithm to Update Gain After a Move

4.3. Implementation Details

Recursive partitioning using FM with and without MCS was run on several designs. The total estimated wirelengths for layouts produced by both algorithms are shown in Figure 4-8. A comparison of runtimes is given in Figure 4-9.

	Description of Design	Total number of elements	Min-Cut	MCS
Design-1	Vax-780 Data Path board	88	1598	1416
Design-2	UART	276	2676	2400
Design-3	Vax-780 Data Path board triplicated	264	5410	4782

Figure 4-8: Estimated Wirelength for Placements Generated by FM and MCS

	Min-Cut/MCS
Design-1	1.15
Design-2	1.10
Design-3	1.12

Figure 4-9: Runtime Ratios for MCS/Min-Cut on a μVax

5. Conclusions

Fundamental problems with the adaptation of MP to do MPLA have been pinpointed. In addition, the Global Effect Problem has been introduced in the context of MPLA. The complexity of the approach has been tested by adapting techniques used in [3] to solve a simplified Global Effect Problem in linear time. Experience with MCS is being applied to develop an algorithm for *globally partitioning* all supercells at a level.

Finally, an algorithm, MCS, has been presented. It may be used to supplement existing techniques such as the use of dummy nodes to improve quality of placement. For example, MCS would complement the use of dummynodes for the situation given in Figure 2-2.

Acknowledgement: This research was partially funded by the CMU-CAD Industrial Affiliates Program and the National Science Foundation under contract #DMC8405136.

References

[1] Burstein, Michael, and Richard Pelavin, "Hierarchical VLSI Layout: Simultaneous Placement and Wiring of Gate Arrays", *VLSI '83*, Vol. , 1983.

[2] Dunlop, Alfred E., "A Procedure for Placement of Standar-Cell VLSI Circuits", *Transactions on Computer-Aided Design*, Vol. CAD-4, 1985, pp. 94-98.

[3] Fiduccia, C.M., and R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions", *19th Design Automation Conference*, Vol. , 1982.

[4] Krishnamurthy, Balakrishnan, "An Improved Min-cut Algorithm for Partitioning VLSI Networks", *IEEE Transactions on Computers*, Vol. C-33, May 1984, pp. 438-446.

[5] Lapotin, David, *Mason: A Global Floor-Planning Tool*, PhD dissertation, Carnegie-Mellon University, 1985.

[6] Lee, C. Y., "An Algorithm for Path Connection and its Applications", *IRE Transactions on Electronics Computers*, Vol. EC-10, March 1961, pp. 346-365.