VLSI Circuit Placement

Implementation using Breuer Algorithm in C++

Rafal Piersiak
Stony Brook University

***Abstract*—Placement is the most important element in physical circuit design. A great placement ensures that area is minimized, and the circuit is fast. A poor placement will not only consume a large amount of area and degrade performance, but it may also cause the placement of blocks to be unrouteable. This could be a very time consuming and expensive problem, which may require another placement. Therefore it stands to reason that a high quality placement will allow for a high quality routing, reducing area and boosting performance. This paper will present the implementation of the Breuer algorithm in C++, discuss the performance of the code, and the resulting output data.**

Keywords— VLSI; Breuer; Placement; Partitioning Based Placement; Quadrature; Bisecting; Slice Bisecting;

I. INTRODUCTION

Placement of cells in a VLSI circuit is considered the most important aspect of VLSI design, due to the number of variables that are controlled by the execution of a high quality placement. Without a good placement, the circuit area will be larger, slower in terms of speed and critical path delay, and may lead to an unrouteable circuit.

The placement of blocks can be considered at three different levels [3]. System level placement is a high level abstract placement of printed circuit boards (PCBs), which minimizes area and ensures efficient dissipation of heat. Board level placement allocates integrated circuits (ICs), discrete components, and other solid state devices to specific locations on a PCB. This placement level attempts to minimize the number of routing layers and allocates devices in a way that maximizes system performance requirements [3]. The quality of placement does not have to be optimal in order to guarantee ability to route, due to a "limitless" amount of potential routing layers. Chip level placement attempts to assign cells to a fixed location in a layout that minimizes the number of routing layers and achieves the required performance specifications. A bad placement, which is found during an attempted routing of cells, can be a major setback, due to a limit on the number of routing layers available for that VLSI design. Because routability can't be verified during placement, the space between blocks is padded to offset potential routing issues. These methods make the circuits larger and a bit slower, but prevent potentially bad placements. To offset the additional wire length, the layout can be compacted later on to lessen the padding effects.

Placement is performed by placing electrical circuits in "blocks," placing such blocks on a sized layout, and ensuring each net can be routed. Total delay is minimized by reducing the critical path in the placement to a reasonable level. Layout area is also reduced, which may decrease the delay in various parts of the placement. Routability can be assessed using a rat's nest, a topological display of nets, which approximates the total wire length as a Euclidean distance between the locations of two pins found on the cells. A good layout will have a rat's nest that minimizes the number of crossing between nets.

The implementation of this concept can be described in mathematical form such that [3]:

- Each block can be placed in a rectangle

  $$R_i\ (w_i, h_i)$$

- No two rectangles overlap

  $$R_i \cap R_j = \emptyset,\ 1 \leq \mathrm{i},\ \mathrm{j} \leq \mathrm{n} \quad (1)$$

- Placement is routable

  $$Q_j, 1 \leq j \leq \mathrm{k} \quad (2)$$

- Total area of the rectangle bounding $\mathcal{R}$ and Q
- Total length is minimized

  $$\sum_{i=1}^{m} L_i \quad (3)$$

- Length of longest net max is minimized

  $$\{L_i \mid i = 1, \ldots, m\} \quad (4)$$

II. RELATED WORK

There are four prominent algorithm classes that work to solve the placement problem. Depending on the inputs to the algorithm, what is the nature of the output of the algorithm, and the process used by the algorithm, the placement algorithms fall into either Simulation Based, Partitioning Based, Other, or Performance Driven Placement Algorithms. All four placement algorithm classifications will be described, along with their respective advantages/disadvantages.

A. Simulation Based Placement Algorithms

This class of placement algorithms simulates some nature process or phenomenon to construct a placement of blocks. Some examples are the arrangement of molecules and atoms to form compact structures with compensated forces and the movement of a heard of animals, which move around to increase the spacing between each other [3]. Three simulation based placement algorithms will be discussed in the following subsection.

1) Simulated Annealing

Simulated annealing simulates the process used to temper metal [3]. This algorithm is an iterative improvement algorithm, which denotes that an initial random placement of blocks is

altered iteratively until the cost of each block to every other block does not create additional swaps based on the probability of performing a cost increase movement using the value of temperature T, which decreases with more iterations. The major advantage of this algorithm is its ability to produce a good placement in a relatively short time [3]. Although it takes some tweaking of the number of perturbations per iteration, final temperature, and cooling rate, the final output is considered high quality and most likely is routable. A major drawback is its ability to get stuck at local optima, which is compensated by selecting appropriate cooling rates. Another common drawback is overlap of blocks, which is not allowed in placement. Fixing overlapping blocks is time consuming, more so than the actual placement algorithm, thus causing any computational benefits from the placement to disappear. These two drawbacks unfortunately push simulated annealing into the realm of small and medium sized VLSI circuits.

2) Simulated Evolution

Simulated evolution simulates the process of mutation of species as they evolve to better adapt to their environment [3]. This algorithm is an iterative improvement algorithm and has the inherent advantage of not becoming stuck in local optima. The algorithm starts with an initial set of placement configurations, possibly randomized, to create a population. During each iteration (generation) the individuals in the population go through a series of fitness tests (mutation, crossover, and inversion) and a set of parents and offspring are transferred to the next generation. This weeds out the weak individuals, although they may mix in with a very low probability. This algorithm will produce a good placement that is routable, but is computationally intense. But simulated evolution is considered better than simulated annealing because it has the added benefit of being able to take previous placement history into account, which increases the likelihood of having an optimal placement. This "history" has a drawback in that it takes much more storage space compared to simulated annealing.

3) Force Directed Placement

Force directed placement utilizes the classical mechanics problem of a system of bodies attached to springs. The force of attraction can be described as the connection of nets between different blocks pulling the blocks towards themselves. The force of repulsion is described as the repulsion of unconnected block. By calculating the distance of nets between different blocks the force if each net on the block is converted to a spring with spring constant k. Longer distances mean greater attraction. The forces of attraction and repulsion are analyzed until the entire system of springs (nets) is at equilibrium. This algorithm can produce good placement at great speed compared to simulated annealing and evolution [3]. It also applicable to general designs, such as full custom where the entire layout is completely customized through specific layer masks during fabrication [3].

4) Other Simulation Based Placement Algorithms

Although the sequence-pair technique is considered a simulation based algorithm, it's coverage in the course textbook and during lecture was minimal and won't be described any further in this paper.

B. Partitioning Based Placement Algorithms

This class of algorithms performs successive placements of cells in blocks, until each cell is inside one block, where cell Ci is placed in some block Bj and the total number of cells is identical to the total number of blocks.

$$\sum_{i=0}^{n} C_i \equiv \sum_{j=0}^{n} B_j \tag{5}$$

The exact method of placements is deeply rooted to the idea of partitioning. A block is cut using both vertical and horizontal lines, until each block contains one cell and each cell is minimally connected to each other cell. This idea is carried out by the Breuer algorithm.

1) Breuer's Algorithm

M.A. Breuer introduced his placement algorithm in a paper, titled "A Class of Min-Cut Placement Algorithms." The Breuer Algorithm was first published in 1977, and presented at the 14th Design Automation Conference.

Breuer presents that previous ideas involving optimizing distance are archaic and a need for new objective functions based upon signal cut is necessary. In his paper, he mentions the three objective functions, Classic Objective Function, Min-Max Objective Function, and the Sequential Objective Function. Breuer also discusses two min-cut algorithms called Quadrature and Slice/Bisecting.

The first objective function (Classic Objective Function) positions that all nets cut by vertical and horizontal lines are summed together.

$$N_c(\sigma) = \sum v(c) \tag{6}$$

Breuer propositioned that minimizing the objective function (6) is equivalent to minimizing the sum taken over all signal nets [2]. He proceeded to show that there are existing methods to achieve this effect.

The second objective function (Min-Max Objective Function) attempts to minimize the number of nets in a channel, with efforts to reduce the maximize track size of that channel as shown in the following equation, where C is a set of cut lines.

$$N_c(mM) = \min(Max\{v(c) \mid c \in C\}) \tag{7}$$

Breuer, himself mentioned that this is difficult to satisfy and is not covered in his paper [2]. The reduction in congestion offered by the Min-Max Objective Function occurs at the expense of potentially having to route nets through sparser channels that increase the net length.

The third objective function (Sequential Objective Function) aims to produce near minimum net cuts by implementing an ordered sequence of cut lines, which is easier to implement. Each cutline partitions a set of cells with a minimum number of cut nets.

The following subsections will show different implementations of sequenced cut lines. Breuer's Quadrature

and Slice/Bisecting placement procedures as well as the Cut Oriented Min-Cut placement procedure will be discussed in detail along with figures displaying the cutlines produced by each procedure.

a) Cut Oriented Min-Cut Placement

This placement procedure cuts blocks and places partitioned cells into these blocks using an alternating sequence of vertical and horizontal cuts. Although this is the easiest placement procedure to implement, the blocks created by the previous cutlines have to be partitioned simultaneously and the number of blocks that need to be placed into a partition may be larger than the partition can allow.

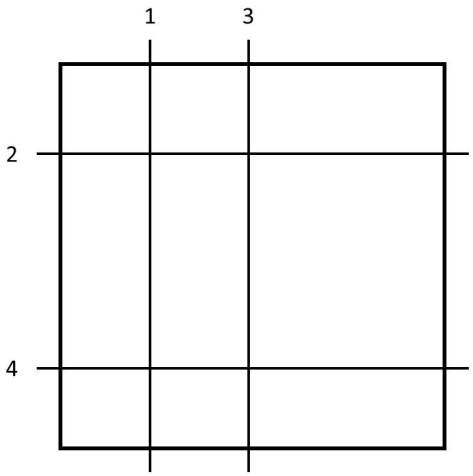

Figure 1: Cut Oriented Min-Cut Placement [3]

b) Quadrature Placement

This placement procedure produces an alternating sequence of vertical and horizontal cutlines which initially start in the center. The cutlines then progress further outwards. This procedure promotes better routability in the center of the circuit, by pushing interconnects outward towards the edges of the layout area. Such a procedure ensures routing is of a uniform density. Quadrature placement is the most popular sequence of cutlines [3].

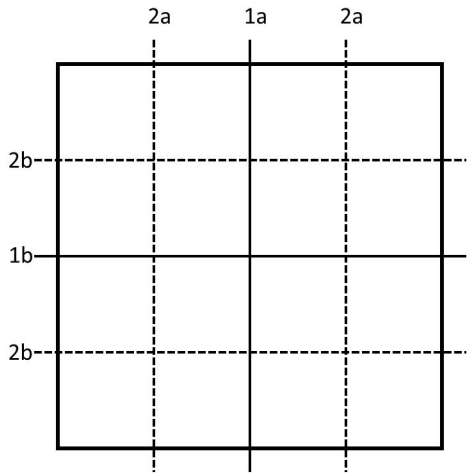

Figure 2: Quadrature Placement [3]

c) Bisection Placement

This placement procedure first begins by bisecting the layout area with a horizontal line through the middle. It then proceeds to bisect the layout further away from the original cutline, ensuring that the same amount of blocks is found on each side of the cutline [2]. This process continues until each sub region consists of one row [3]. The same procedure is performed using vertical cutlines, until each cell is located inside a block. Bisecting doesn't necessarily minimize the maximum net cut per channel.

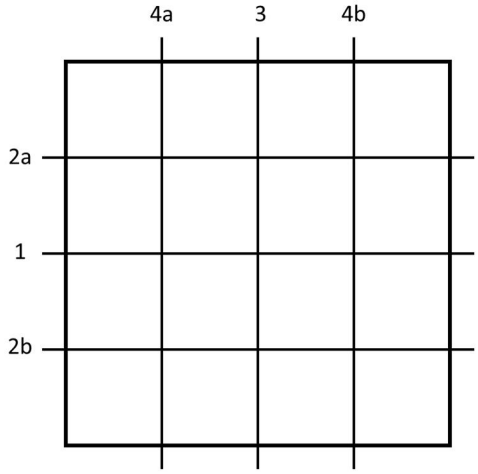

Figure 3: Bisection Placement [3]

d) Slice Bisection Placement

This placement procedure performs an iterative horizontal slicing of the layout area. Each horizontal cut contains a predefined number of blocks placed above the cutline. The horizontal cuts continues downward until all blocks are assigned to a row. Then the layout is bisected with vertical lines, starting with an initial vertical cut in the center and proceeding cuts that ensure equal amount of blocks on either side of the vertical cut line. This procedure is best suited to minimize the number of interconnects at the periphery of the layout.

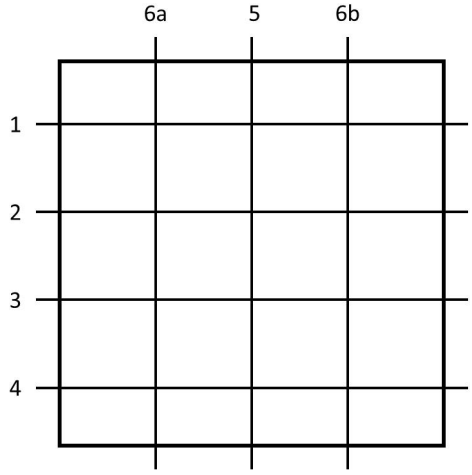

Figure 4: Slice Bisection Placement [3]

2) Terminal Propagation Algorithm

The terminal propagation algorithm is an extension to placement based algorithms. A typical placement algorithm such as Breuer's algorithm does not take into account interconnections between cutlines already present in a layout area. This can lead to cells that end up being further away from each other, causing an unnecessary increase in net length and increased congestion in channels [3].

By creating dummy terminals between terminals that have been cut by a cutline, a partitioning based placement algorithm will know not to move these dummy terminals to partitions that will require routing through two or more blocks. This implementation keeps terminals close together and minimizes the length of certain terminals separated by a cutline. The following figure compares the placement of two cells using a typical partitioning based algorithm, to one that utilizes the terminal propagation algorithm along with a typical partitioning based algorithm.

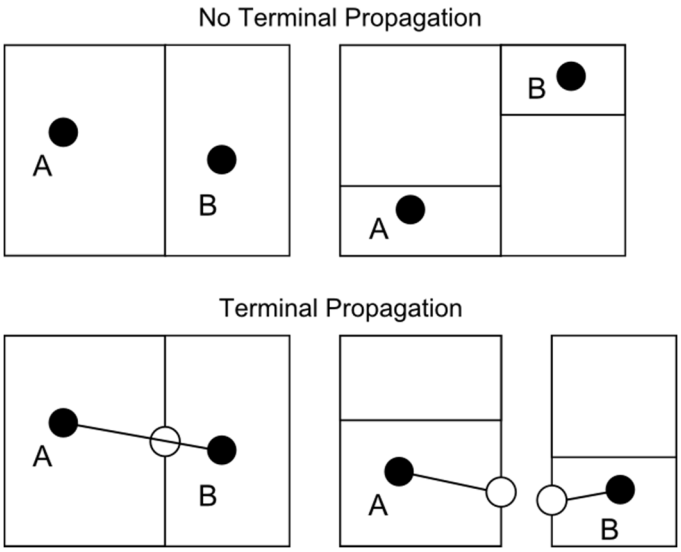

Figure 5: Comparison of Partitioning Based Algorithms With and Without Terminal Propagation [3]

C. Other Placement Algorithms

This class of algorithms does not depend on natural phenomenon or partitioning algorithms to create a placement solution. In this section: Cluster Growth, Quadratic Assignment, Resistive Network Optimization, and Branch-and-Bound Technique will be examined briefly.

1) Cluster Growth

The cluster growth algorithm is a degenerate placement algorithm, in that it is used primarily to assist iterative algorithms which depend on an initial placement of blocks before running. It works by placing a block selected by the user, and then finding the block that is the most connected to that first block. The algorithm places that highly connected block as close as possible to the previous block. Because this algorithm does not take into account interconnections between other blocks and other circuit features, it doesn't produce a good placement [3].

2) Quadratic Assignment

The quadratic assignment algorithm is a purposed algorithm for gate array placement and therefore not widely applicable to other design styles. The algorithm creates a connectivity matrix and attempts to map the blocks into slots with minimum overall cost, which is equivalent to reducing the total wire length for the circuit [3].

3) Resistive Network Optimization

The resistive network optimization algorithm transforms this placement problem into a resistive network intent on minimizing power dissipation. It has many advantages, including allowing irregular sized blocks within cell rows, efficiency of placement, and slot constraints maintaining legality of placements. It performs a placement by representing pads and fixed as fixed voltage sources and performs the steps optimization, scaling, relaxation, partitioning, and assignment to align blocks to their slot locations [3].

4) Branch-and-Bound Technique

The branch-and-bound technique simulates movement through a tree, which is pruned to decrease computation time. The algorithm starts with a block B1, and traverses a branch. The connected slot node is calculated for total wire length at placement. If this cost is lower than the previous value, the algorithm continues moving downward towards the next block to place the next slot. If the succeeding slot cost is higher than the lowest cost, the algorithm performs a bound step and searches another part of the tree. This continues until all slots have been placed in the blocks. Although this methods performs some shortcuts to limit searching all possible placement configurations, it is computationally expensive and is relegated to partitioning only small circuits [3].

D. Performance Driven Placement Algorithms

This class of algorithms places blocks on the presumption that minimizing the wire length will minimize delay in all interconnects. One method places the blocks in order keep the path length between blocks within timing constraints. The net-based approach tries to route nets in a way that meets that individual nets timing requirements. This method would require a pre-timing analysis of the nets in order for the algorithm to know what the individual net timing constraints are. The algorithm depends on a set of upper bounds of timing constraints for the nets and utilizes a modified version of Feduccia's min-cut placement algorithm to minimize the number of nets exceeding the upper bounds while limiting cutsize. If the timing constraints for all the nets are not met, the upper bounds are altered and the algorithm performs a revised placement. This is the only placement algorithm that generates placements with timing constraints [3].

III. PROPOSED SOLUTION

The implementation Breuer's algorithm described in this paper is written in C++ using the C++11 coding standard. The IDE and compiler used for the construction and running of this program is Visual Studio Ultimate 2013. The program consists of a main file and nine secondary files that contain the functions and variables called out in the main program. Each secondary file has its own header file, bringing the total file

count for this program to 19 files. This was done to aid readability and provide a formal structure to the program.

The initial iteration of the program can be described in the following manner:

1. Parse input .nodes file
2. Parse input .nets file
3. Run Breuer algorithm
   a) Randomize initial two partitions
   b) Perform Kernigan-Lin
   c) Move partitioned elements to blocks vector
   d) Repeat b-c until all blocks have one element
4. Output blocks to a text file
5. Run placement imager program

The parse nodes function takes in the input .nodes file and count the number of nodes and terminals in the circuit. Terminals are considered pads, which are not factored in during parsing. As the file is read, each node has an associated width and height associated with it. The array produced by this function creates an N x 2 array, where N is the total amount of cells in the file. The index acts as the node number and the two variables in each row are the width and height.

The parse nets function takes in the input .nets file and creates a 2D vector with N rows, where N is the total amount of cells in the file. As the file is read, each source node has its associated sink connections inserted into the respective row of the 2D vector. The size of each row is equal to the degree of the vertex. During parsing, the .nets file only specifies an edge in one direction. For example, if vertex 0 is connected to vertex 4, an entry will be pushed into row 0, with the value of 4. But this only represents half of the information. To correct for this misrepresentation, an entry is placed in row 4, with a value of 0. This ensures that for any vertex, we know each other vertex it is connected to. The parse nets function also recovers the pins associated with each connection between source and sink nodes. These pin coordinates are stored in an array N x (#connections * 4). The number of connections is multiplied by four because each time a sink node is encountered the pins for the source are also copied over, plus each node has two pins.

The randomize function creates a sorted vector with entries from 0 to N – 1. The shuffle function is called to randomize the entries in this vector. Half of this vector is copied into a vector designated as partition A, and the other half is copied into a vector designated as partition B. The randomize function is only run one time, during the initial setup before the Kernigan-Lin algorithm is run for the first time. Otherwise, the randomize function is not used.

The generateG1 function creates an N x N + 4 array, which contains the D values for all he vertices. Using an N x N array facilitates the ease of calling a vertex[i][j]. The addition of the extra 4 columns is used to store whether the vertex is in partition A or B, the amount of inside and outside edges, and the D gain of the vertex. The 2D vector contains one of four values that signify, whether the source vertex (row) is in A or B and whether its sink vertex (column) is in A or B. This value is used to determine how to compute the D gain for that source vertex. During initialization, all vertices have their D gain values updated. Once the next iteration takes place, only the D gain values of the swapped vertices is altered, and all respective sink vertices associated with them. This eliminates a major chunk of time that could be wasted. The function exits and returns the initial cut size during the initial run. The returned value is not used for subsequent runs.

The generateG2 function calculates the G gain values for all the possible combinations, N^2. If there are four vertices in each partition, the G gain will be calculated 16 times. The function reads the value at vertex[i][j] along with the D gain of vertex I and vertex j and calculates the G gain. The function exits and returns a struct containing the max G and the two vertices responsible for the gain.

The kerniganLin function combines the previously mentioned files to perform partitioning of a set of nodes into two partitions. The program returns a 2-row vector with X nodes per row. The first partition are stored in first row of the array, while the nodes from the second partition are stored in the second row of the array. This can be run multiple times to partition a large circuit into smaller partitions.

The breuer function manages the 2-D blocks array, which contains a cell in each location in the blocks array. It performs many Kernigan-Lin partitioning iterations to push all the cells of the circuit into a block of size one. The algorithm also deals with controlling how vertical and horizontal cuts are made and prepares the partitioning arrays for the Kernigan-Lin algorithm.

An output file generates a list of blocks with the (Node Number, X, Y, Width, Height) in each row in the file. The first line states the size of the layout area (Width and Height). The file terminates by writing “End” on the last line.

This file is fed into an imaging program written in C# that has been compiled as an executable. The blocks.txt file and the CSharpBitmap.exe are placed in the same directory. The executable is run and a window pops up with the respective placement. The cells are represented as various sized rectangles. Each rectangle has a number inside the rectangle, which denotes the node number. The executable also outputs a png image filed labeled “placement.png.” Therefore the user can view the output of the placement even after the program is closed.

A. Relevant Implementation Issues of Proposed Solution

The parsing speed of the .nodes and .nets file is almost instantaneous. All vertices are assigned their respective connections to other vertices quickly with minimal storage. The parsing function can be coded to remove the creation of a 1D vector that stores the number of connections each vertex contains, because the size of the row of the 2D vector can be utilized for that purpose. The pare functions also save pin coordinates and node width and height information.

The main improvement to the partitioning algorithm is more of a compiling improvement rather than any altered code. I was not aware that you can make a “Release” version of your program. This is usually done to minimize the executable file size and to allow code optimizations to quicken the code. I was running my partitioning program using the “Debug” mode which produces a lot of additional code to all you to watch

variables and initiate breakpoints, among other things. By building a "Release" version of my partitioning program, I was able to run the program in less than 30 minutes for ibm01.nets/ ibm01.nodes, which contains 12506 cells. This is a major speed improvement compared to over two hours using the "Debug" build.

I had many issues with creating the Breuer algorithm. I was able to use the algorithm to partition four nodes into four blocks and create a pseudo layout. I did not factor in spacing between blocks, channel width, layout area, etc. Therefore the output blocks.txt contains hardcoded coordinates for the cells. The width and height of the cells are taken straight from the nodes file.

The most difficult task is to figure out how to split up the partitions and place them into blocks. A method was developed on paper, but implementing this method of shifting cells inside blocks is quite difficult and very frustrating. The difficulty also comes about due to the idea of implementing vertical and horizontal cuts. Cutting the layout containing blocks containing nodes was daunting and was only developed to perform one vertical cut and one horizontal cut for a total of four nodes. Two perform Breuer's algorithm on more than four nodes would require additional coding which manages the partitions, nodes, blocks, and cutlines dynamically. A technique found in the figure below was developed to shift blocks and nodes around, but it proved to be very difficult to program.

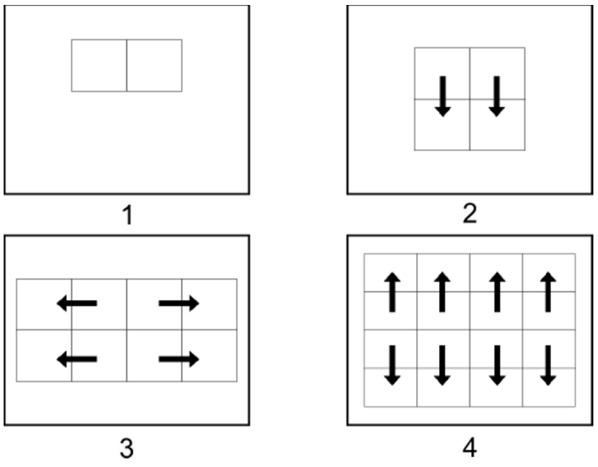

Figure 6: Dynamic Movement of Blocks with Nodes Inside

Another issue with the Breuer algorithm is due to the use of the Kernigan-Lin (KL) algorithm. The KL algorithm requires equal partitions, so each time a set of blocks are portioned, before any blocks can be portioned again, the partitions must be checked if they are even. If they aren't, a dummy node needs to be added. If the Breuer algorithm in this paper was done correctly, the layout placement would contain a bunch or open areas where the dummy nodes would go. This is an inefficient use of area. Due to being tasked with using the KL algorithm for the previous project, my ability to program the Breuer algorithm became more difficult, compared to if the Fiduccia-Mattheyses algorithm was used.

Another time consuming task was trying to get the Kernigan-Lin algorithm to partition nodes multiple times. The code was written in such a way that didn't make it possible to reuse multiple times. I had to rewrite a decent portion of the code to be able to partition smaller and smaller sets of nodes. Also the algorithm will most likely still not be able to partition files larger than the ibm01 file because the same two massive arrays are still used to calculate the gain values.

The blocks.txt created by the Breuer algorithm has some formatting issues, which cause it to not work with the imager program. A text filed labeled goodplacement.txt will be included in the archived package. It can be renamed as blocks.txt and placed in the same directory as the CSharpBitmap.exe file. The output of the program is found in the following figure.

The creation of the CSharpBitmap executable was surprisingly considering I never used C# before. I was able to create this program in about six hours. I mainly started work on this program due to severe frustration with figuring out how to manage the nodes inside the blocks.

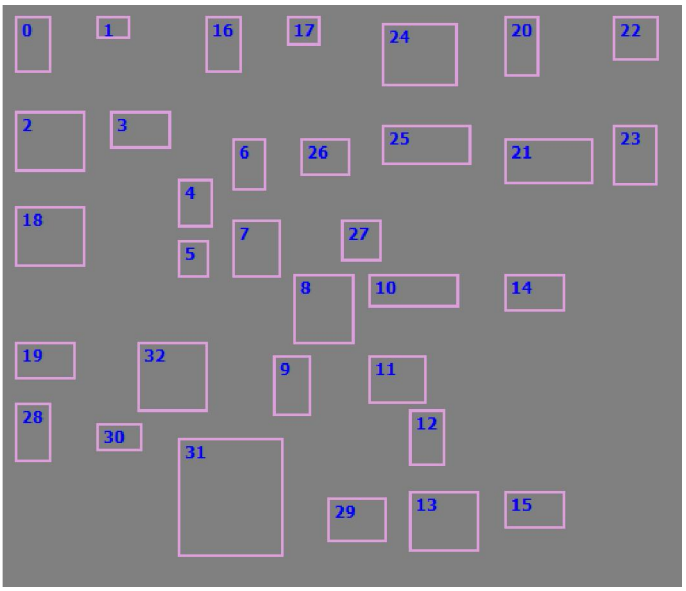

Figure 7: Good Placement Layout Created with CSharpBitmap.exe

IV. Experimental Results

The nodes and nets files tested during the creation of the Breuer algorithm were found in [1] and [4]. Also a number of other files were used found on the UCSan Diego VLSI page titled "Small Circuit Partitioning Instances." These files allowed me to test my partitioning algorithm much quicker because execution time was almost instant.

The only experimental results that were very successful is the generation of the layout using a placement imager program I created. The Breuer algorithm was only able to assign four nodes to one block. The algorithm is not able to process more than four nodes due to difficulty in writing the additional lines that would dynamically manage the blocks and nodes no matter how many nodes were present in the nets and nodes files.

The overall program outputs a text file contain the layout area and the cells with location and size, which can be read by the placement imager program.

V. Conclusion

An attempt to reproduce the Breuer algorithm for this placement project was primitive to say the least. The basic concept of allocation one node to one block in a layout area was successfully carried out with up to four nodes. Weaknesses in control and data structures made attempts at creating an algorithm that can handle more than four nodes difficult.

Acknowledgment

This paper on the implementation of the Breuer algorithm in C++ was submitted as a project paper for ESE 556 - VLSI Physical and Logic Design Automation, conducted by Professor Alex Doboli at Stony Brook University.

References

[1] A. Caldwell, A. Kahng, and I. Markov. (2013, Nov 17). "Small Partitioning Instances." Online.Available: http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/Partitioning/smallPP.html

[2] M. A. Breuer (1977), "A class of min-cut placement algorithms," Proceedings of the 14th Design Automation Conference.

[3] N.A Sherwani "Placement," in Algorithms for *VLSI Physical Design Automation*, 3rd ed. Norwell, K.A.P., 1999, ch.7

[4] X. Yang. (2013, Nov 17). "Dragon: A Placement Tool Generating Highly Routable Designs."Online. Available: http://er.cs.ucla.edu/benchmarks/ibm-place2/