THE UNIVERSITY OF TEXAS AT DALLAS

Erik Jonsson School of Engineering and Computer Science

Fiduccia-Mattheyeses based graph partitioning

EEDG 6375 : Design Automation of VLSI Systems

Submitted by,

Akash Anil Tadmare (aat171030)

Akshay Nandkumar Patil (anp170330)

Date,

December 8, 2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction:

VLSI world completely revolves around integrated circuits (IC). Manufacturing process of an IC follows through a lot of different steps, out of which typically IC design cycle involves following steps:
1. System Specification.
2. Architectural Design.
3. Logic Design.
4. Logic Synthesis.
5. Physical Design.
6. Physical verification and Tapeout.
7. Wafer Fabrication.
8. Packaging.

Amongst all these steps, we will be dealing with Physical Design. Once the logical description of the design has been completely synthesized into a netlist, the physical design converts this netlist based circuit representation into a geometrical representation.

Physical design further involves following steps:
1. Partitioning.
2. Floor-planning.
3. Placement.
4. Clock-tree synthesis.
5. Routing.
6. Timing verification.

It would be very time consuming or even impossible to hand-craft the circuit design of large complexity without any assistance of computer programs during all the steps discussed above. According to Moore's law, number of transistors on a single chip doubles every 2 years, the state of the art chips consists of an entire system with millions of transistors. For example, Pentium processors from Intel has 3 million transistors. Computer programs are used widely into the industries to automate these physical design processes with no or very little human intervention. As a part of Design automation of VLSI systems course. We have implemented automatic partitioning and placement engines.

1.1 Importance of Partitioning:

Partitioning is the process of dividing a chip into different functional blocks and aim is to keep the frequently communicating blocks together. This reduces the cost and efforts of further design stages. In case of FPGA based design, when the design is too large to map on a single FPGA board, the circuit is partitioned and multiple blocks are mapped on different boards. The aim of the partitioning process is to divide the design in such a way that, the interconnection between the blocks are minimized. If the interconnections between two partitions are not minimized, then the functional blocks which are supposed to be together will be pushed away from each other which will affect the overall performance of the system. In FPGA based implementation, the interconnections between the boards will increase thereby increasing the complexity of interfacing the boards together.

The partitioning problem takes in a netlist and divides the circuit represented by the netlist into two parts with a target to minimize the interconnection between them.

2. Circuit Partitioning using Fiduccia and Mattheyeses algorithm:

In the world of VLSI Design automation, the circuit partitioning is the most fundamental step that is followed during the design of the backend of the EDA tools. With the help of partitioning, the circuit is divided into two parts in such a way that the cells which interact together more frequently are kept close by i.e. in the same partition. In other words, the number of nets across the boundary of cut is minimized. For partitioning purpose, an electronic circuit is converted into graph-based model G(V,E). A graph partitioning method was proposed by Kernighan-Lin (KL method) in 1970. In this method nodes from two partitions are swapped with each other and the swap which increases the current cutset between two partitions is rejected. This was proven to be effective for circuits with relatively smaller solution space but it gets stuck in local minima for the bigger circuits. Another algorithm especially for circuit partitioning was proposed by Fiduccia Mattheyeses in 1980 known as FM method. The FM method is an improvement over KL method in terms of both, execution time and final cut-set. Unlike KL method which makes the move only if there is an improvement, FM method encourages to make some bad moves in a hope that this will lead to a better solution (cut-set) in the future also known as Hill-Climbing. FM method proved to be the most effective method for circuit partitioning also, it makes use of efficient data structures to avoid unnecessary searching for the best cell to move and to minimize unnecessary updating of the cells affected by each move.

2.1 Details of FM algorithm:

Conversion of a circuit into graph model:
First, we convert the circuit into Hypergraph-based model G(V,E) as shown in the figure1
Where,
V: Set of cells in the circuit. {c1, c2, c3, c4, c5}
E: Set of interconnect in the circuit. {n1, n2, n3, n4, n5}
Let,
n(i) be the number of cells on net ni,
s(i) be the size of cell ci,
p(i) be the no. of pins on cell ci,

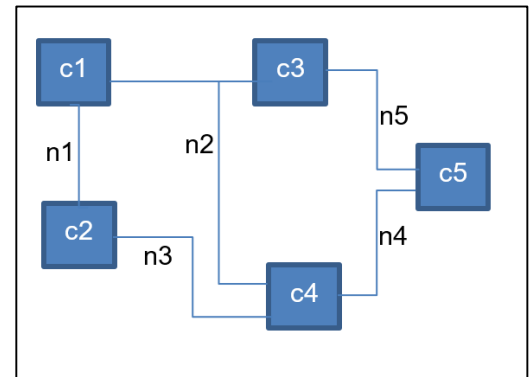Total number of pins in the circuit $P = \sum_{i=1}^{cells} p(i)$


Figure 1: Hypergraph model of a circuit

2.1.1 Random partition and initial cutset:

First step is to partition the circuit into random two parts such that it satisfies balance criteria. Figure 2 illustrates this on an example circuit. The concept of partitioning for minimum cutset is meaningless unless we put some restriction for the partition otherwise we could achieve an empty cutset by moving all the cells in one partition. In FM this restriction is implied with the ratio "r" for which,
Let s(A) and s(B) be the total sizes of part A and B respectively.
$r \approx \frac{S(A)}{S(A)+S(B)}$, The partition satisfying 0 < r < 1 are accepted.
After random partition the total number of nets crossing the partition boundary becomes the cutset of the circuit.
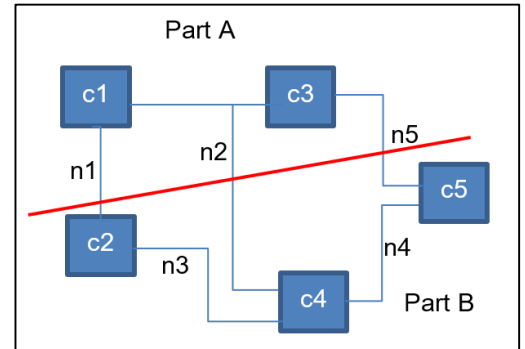
2.1.2 Gain calculation:

Now we start making the moves towards minimizing this cutset.
The moves are based on the gain of a cell which is defined as, the change in cutset when the cell moves from its initial partition to the other partition.
For all nets connected to cell $c(i)$, the nets crossing the partition boundary are called external nets "E", and remaining nets are internal nets "I" and gain $g(i) = E-I$.
 e.g. For cell c4, E=1 and I=2 as, n2 is an external net and n3 and n4 are internal nets.

$$g(4) = 1 - 2 = -1$$

The gain is negative/positive if the cutset increases/decreases after the move. The maximum gain of a cell $c(i)$ can be $+p(i)$ and minimum can be $-p(i)$.

2.1.3 The Bucket Array:

Once the initial random partitioning of the cells is done, their initial gains are calculated. An array type data structure is used to create two buckets, bucketA and bucketB one each for both partitions. Buckets store the list of cells with same gain as shown in the figure 3.
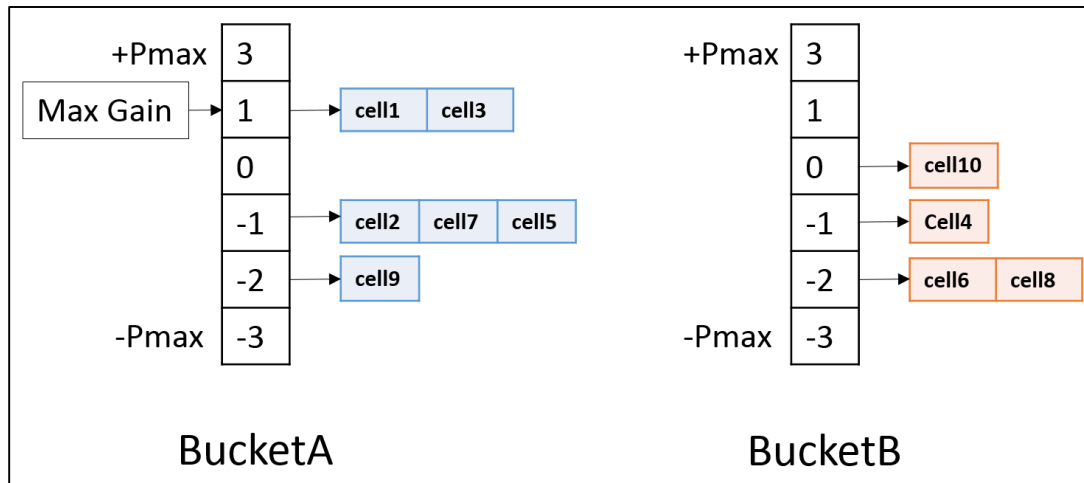
Figure 3: The Bucket Array

A base cell of maximum gain is selected to move and along with buckets, a list of locked cells is kept. Whenever the base cell is moved from one partition to another, that cell is added to the locked list and is removed from the bucket list. Once it is moved to complimentary partition, the gains of all neighbors of the base cell may change. Their gains are updated along with their respective buckets. when no cells are left with a positive gain, then a cell with negative gain is still moved in a hope that it will improve the cutset after the move.
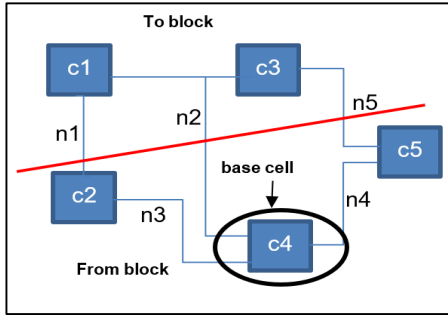
Figure 4: Illustration of 'From' and 'To' blocks

2.1.4 Fast computation of gains:

The gain calculation discussed previously is a naive approach and take a lot of time when it comes to updating the gains after the move. To speed up this process a new method is proposed.

Let, F be the current (from) block of the base cell and T be the complimentary (to) block as shown in figure 4. For every net 'n' connected to the base cell $F(n)$ and $T(n)$ be the number of cells the net 'n' has in the 'from' and 'to' blocks respectively. The net participates in the cutset if $T(n)$ is non-zero and the net is critical before the move if $T(n)=0$ or 1 or $F(n) = 1$ which is base cell. Similarly, the net is critical after the move if $T(n)= 1$ or $F(n)= 0$ or 1. While updating the gains of the neighbors of the base cell, all these conditions are checked, and the gains of the respective cells are incremented or decremented according to the conditions. Updating the gains in this way saves a lot of time completing the entire process in linear time.

2.1.5 End criteria:

The process of moving a cell and updating the gains is continued until either all cells have been moved and locked or no more moves are possible. At this point, a pass is said to have completed and the next pass starts from the state where minimum cut was obtained in this pass.

2.2 Implementation Details:

The FM algorithm discussed above was implemented in C++ and tested on the ISPD2016 benchmarks. The details of implementation are discussed below.

2.2.1 Data structures:

As C++ is an object-oriented programming language. It offers object oriented constructs such as classes which can be used to store the parameters related to a specific object. In our implementation we used two classes, namely, cell and net and are described below

1. **Cell**

Table 1: Cell class

Cell Size	Cell Gain	Lock status	Cell partition	Cell type	Netlist
Stores the size of the cell	The gain of the cell is stored in this member	Stores '1' if the cell is locked else stores '0'	Stores '0' if cell is in partition A else stores '-1' for partition B	Stores the type of cell e.g. LUT, DSP, RAM.	List of nets connected to the cell

Table 1 shows the members of the class 'cell'. All cell related data from the cell class can be accessed using an object of that cell. The objects are stored in another construct i.e. map named as cell map. Map stores the data in (key, value) pair where key is a pointer to the value of the map. The data access through map can be done in O(1) time.

4

Table 2: Structure of map of cell

Cell no.(key)	Object of class cell (Value)					
0	Object of cell class corresponding to cell 0					
1	Object of cell class corresponding to cell 1					
2	Cell Size	Cell Gain	Lock status	Cell partition	Cell type	Netlist
3	Object of cell class corresponding to cell 0					
. .	. .					
n	Object of cell class corresponding to cell n					

Table 2 shows the structure of cell map which stores the cell number as key and value is the object of the cell class corresponding to that cell with an example to illustrate how this map will be accessed for cell number 2.

2. Net

Table 3: Net class

Cell List	A size	B size
Stores the list of cells connected to the net	Total number of cells connected to the net from partition A.	Total number of cells connected to the net from partition B.

Table 3 shows the members of the class 'net'. All net related data from the net class can be accessed using an object of that net. The objects are stored in another construct i.e. map named as net map.

Table 4: Structure of map for nets

Net name (key)	Object of class net (Value)		
net_0	Object of cell class corresponding to net 0		
net_1	Object of cell class corresponding to net 1		
net_2	Cell List	A size	B size
net_clk_buff	Object of cell class corresponding to net clock buffer		
. .	. .		
net_i	Object of cell class corresponding to net i		

Table 4 shows the structure of net map which stores the net name as a key and value is the object of the net class corresponding to that net with an example to illustrate how this map will be accessed for net 'net_2'.

3. Buckets A and B:

As discussed in section 2.1.3 we used maps to store the data of buckets. In this map the key is the gain of the cell and value is the list of cells corresponding to that gain. Two separate buckets bucketA and bucketB are used for partition A and B. Keys for the maps are always stored in a sorted order, so the key at the top always points at the list of cells with maximum gain one of which can be selected as a base cell. Again data access in O (1) time speeds up this process.

4. Locked cells:

A vector array is used to keep a list of locked cells. A cell is pushed into this vector when it moves from its partition to the complimentary partition and gets locked. This list also provides the information about the sequence in which the cells were moved and locked. We keep track of the cutset and the index of the cell, which was pushed into this list at the occurrence of minimum cutset, is stored. This index will be used to undo the moves made after mincut to start the next pass.

2.2.2 Implementation:

Implementation starts with reading the nodes file which stores the information of cell number and cell type for all the cells from the circuit. While reading this file, the number and its type is stored in the data structure. At the same time, the cells are pushed into one of the partition randomly. The pseudo code below shows this flow

```
/* cell list file */
FOR each cell c = 1 … N
      store the cell number cell c;
      store cell type for cell c;
      store size of cell c;
      randomly pick a partition for cell c without violating balance criteria;
      unlock cell c;
END for
```

Once the nodes file is done reading, netlist file read. This file contains the names of each net and the cell numbers connected to it. Using this information, the cell list of each net as well as the net list of each cell is updated in the data structure.

```
/* net list file */
FOR each net n = 1 … N
   FOR each cell c on the net n
      update cell list of net n;
      update net list of cell c;

      if (cell c in partition A)
            A size++;
      else if (cell c in partition B)
            B size++;
   END for
END for
```

Now that all the information necessary for gain calculation is available, initial gains for all cells are calculated as follows:

```
/* Initial gain calculation */
FOR each cell  = 1 … N
   FOR each net n in the netlist of cell c
      F <- current partition of cell c (FROM partition)
      T <- complimentary partition of cell c (TO PARTITION)

      if ( F(n) == 1 )
            gain(c)++;
      if ( T(n) == 0)
            gain(c)--;
   END for net list
   update bucket for FROM partition of cell c
END for cell
```

After this, the whole data structure is updated and ready to be used for actual partitioning process. This partitioning process follows the following steps:

```
/* MAIN LOOP*/
WHILE (!exit_criteria)
{
   DO
   {
       Select a base cell to move
   }WHILE ( balance criteria not met )

    remove base cell from current bucket;
    lock base cell;
    push to locked cell list;
    cutset = cutset – gain(base cell);
    change partition of base cell;
    update_gains();
}
```

The `update_gains()` is the function which is called in the main function to update the gains of all neighboring cells to base cell. The update gain routine goes as follows:

```
/* UPDATE GAINS */
F <- current partition of base cell (FROM partition)
T <- complimentary partition of base cell (TO PARTITION)

FOR each net n in the netlist of base cell
   if ( base cell from A )     //assuming base cell is from partition A
      F(n) = A_size(n);
      T(n) = B_size(n);
   /* Check criticality before the move */
   if (T(n) == 0)
      Increment gains of all free cells on net n;
      Update_buckets();
   else if(T(n) == 1)
      Decrement the gain of the only cell (if unlocked) present in T partition
      on net n;
      Update_buckets();

   /* Making the move */
   F(n) --;
   T(n)++;

   /* After the move */
   A_size(n) = F(n);
   B_size(n) = T(n);

   /* Check criticality after the move */
   if(F(n) == 0)
      Decrement gains of all free cells on net n;
      Update_buckets();
   else if (F(n) == 1)
      Increment the gain of the only cell (if unlocked) present in T partition
      on net n;
      Update_buckets();
END for cell
```

Each time when the gains were updated in *`update_gains()`* function, *`update_buckets()`* function is called. It was necessary to reflect the gain change in the bucket arrays. The function takes the cell whose gain has been changed as well as its previous gain as input and updates the bucket arrays accordingly. The flow of *`update_gains()`* goes as follows:

```
/* UPDATE BUCKETS */
F <- current partition of cell whose gain has been changed (FROM partition)

in bucket(F)
Remove the cell from the list pointed by previous gain;
Add the cell to the list pointed by new gain;

/*END UPDATE BUCKETS*/
```

After this the program returns to the main loop and selects next cell to move. This process continues until exit criteria is met.

2.2.3 Exit criteria:

Exit criteria is same as discussed before, i.e. the program ends when no more cells are left unlocked or no more cells can be moved satisfying the balance criteria.

2.3 Testing and results:

By implementing the algorithm strategy discussed above, we created a source code in C++ and simulated on 4 benchmarks. To test our code, we were given four real world FPGA example circuits. These are ISPD (International Symposium on Physical Design) benchmarks which were given in the "*ISPD 2016: Routability-Driven FPGA Placement Contest*". The brief description of these benchmarks is given in the table 5.

Table 5: Characteristics of Benchmarks

| Benchmark name	Size in number of cells	Size in number of nets
FPGA-example 4	844,184	849,984
FPGA-example 3	427,800	430,600
FPGA-example 2	542,239	545,089
FPGA-example 1	3,336	10,037

From table 5, it can be seen that the size of example 1 circuit is very small as compared to the other circuits. Therefore, the number of passes for example 1 circuit can be increased. The benchmark example 1 was simulated for 10 passes and example 2 to 4 were simulated for 5 passes.

The table 6 briefly tabulates the results after doing these simulations

Table 6: Simulation results for FM Partitioning

Benchmark name	Starting Cut	Ending Cut	Percentage change	Execution time per pass
FPGA-example 4	633724	48263	92.38	49min 10s
FPGA-example 3	323135	27140	91.60	14min 37s
FPGA-example 2	386429	34936	90.96	19min 50s
FPGA-example 1	2398	91	96.20	600ms

2.4 Conclusion:
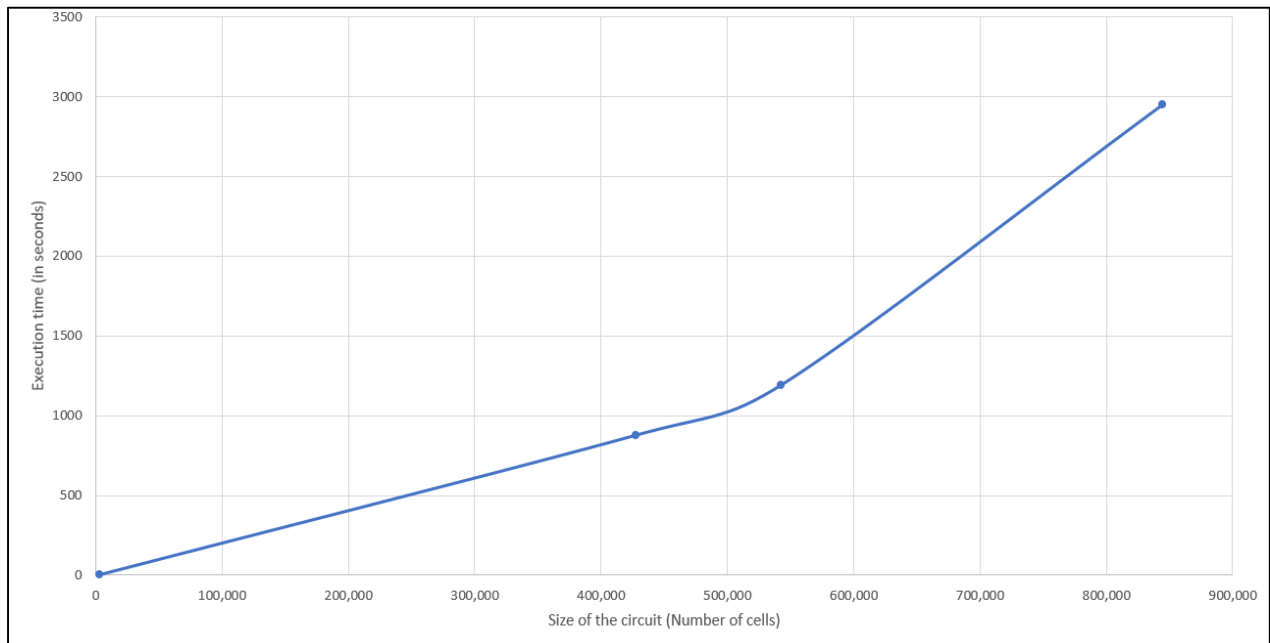
Figure 5: Size of the circuit v/s Execution time

As the circuit size increases, it becomes practically impossible to perform an exhaustive search to get the best acceptable partitioned circuit. The FM algorithm provides a very fast approximation heuristic for mincut partitioning of the circuit based on iterative improvements. The figure 5 shows that the execution time grows linearly with the size of the network, whereas the execution time with the exhaustive search approach may require factorial time.