Abstract - VLSI designs in this age have billions of components. It has become increasingly difficult to design and manipulate circuits approaching these sizes. One workaround for this problem is partitioning. Partitioning is the process of dividing a circuit or system into smaller subdivisions so as to minimize the number of interconnections between the subsystems in order to speed up the subsequent placement and routing. This project is aimed at developing and testing the Simulated annealing algorithm in C++ used for partitioning circuits. The algorithm so coded is tested on several benchmarks and the results obtained are tabulated and plotted.

INTRODUCTION:

Partitioning is an integral step in circuit design. In simple terms, a set of components or modules act as input to the coded algorithm, these sets are then taken and form an output, which resembles the input but with a lot less interconnections as compared to the initial input. This drastically reduces the design complexity and makes the process of placement and routing much more manageable. The use of this technique has its advantages. Some of them include reduction in the number of wires required, the length of the wires required for the interconnections, reduction in the area of the chip thereby reducing the total cost required for the fabrication for the chip. Moreover it reduces the delays which are a significant problem in circuits these days. When designs are extremely large, there is a high probability that modules have to be reused in an entirely different area of the chip. Partitioning reduces the workload by identifying these modules that are required multiple times throughout the chip and thereby constructing functional modules out of these in a manner similar to functions in a programming language. The subdivisions can be designed independently of each other by entirely different teams in order to speed up the design process. Partitioning is of three types: System level, board level and chip level partitioning. System level partitioning is dividing the system into smaller blocks so that smaller blocks can be designed and fabricated on the PCB independently. But if the PCB is too large we go for board level partitioning, where the circuit assigned to a PCB is partitioned into sub circuits so that each circuit can be fabricated as a VLSI chip. At chip level partitioning the circuit assigned to the chip is partitioned into smaller sub circuits. The main aim of partitioning is to reduce the number of nets that cross the partitioned boundaries. This is represented using the term cut size which gives the number of nets connected between multiple partitions. Obtaining a minimum cut size is the primary aim of partitioning. Partitioning can be done by several algorithms including Kernighan-Lin, Simulated Annealing, Fiduccia-Mattheyes, Tabu search and Genetic algorithm. Although there are a lot of greedy algorithms for partitions, simulated annealing works just as efficiently and provides solutions that are comparable to the greedy algorithms

ALGORITHM:

This algorithm tries to reduce the cutset by simulating the annealing process. A sufficiently high start temperature is chosen, and allowed to cool very slowly. At every temperature there are perturbations done on the solution set, which are accepted or rejected based on a function of cost and temperature. The function is defined as follows" *exp ((initial solution- neighbouring solution) / temperature) "*. The solution is accepted or rejected after comparing the function against a random probability (ranged between 0 and 1). Good solutions are automatically accepted .For bad solutions , the temperature acts like a gatekeeper which initially accepts any solution, but only accepts good solutions as it cools

down. The initial solution accepted is iteratively improved by local changes until no better solution is found. Simulated annealing allows for accepting bad solutions so as to not get stuck at locally good but globally bad solutions.
The influence for this algorithm is the actual annealing process in metals where the metal is heated to a high temperature and then cooled down very slowly. If the cooling process is slow enough to ensure thermal equilibrium and the heating temperature is sufficiently high to ensure random state, the atoms of the metal place themselves in a pattern that very closely resembles a perfect crystal. The algorithm can be summarized by the following steps:
Intialize - Random initial placement. High temperature initialized.
Disturb/Move - Disturb the placement by moving a cell.
Calculate cost- Due to the move made in the previous step, calculate the cost factor.
Select - Based on the cost factor or function, accept or reject the move made in step. There is a probability of acceptance here.
Update - The temperature is lowered and the algorithm is repeated from the Disturb/Move step. This is done until lowest temperature is reached in a manner similar to that of the simulated annealing process in metals.
We have considered the following parameters in designing algorithm:
Delta change in area
5% .Movement of cells is restricted such that each partition has an area equal to half the total area with an error of 5%
Start temperature –
After the initial random partition, every cell in partition A is allowed to swap with a corresponding cell in partition B. For this initial partition, the best and worst case values of the cutsets are stored. The start temperature is chosen with the following assumption:
"Assuming a very good initial cutset, the start temperature should accept a worst case cutset if the probability = 0.8"
The value of 0.8 was arrived at after performing several iterations on one benchmark and checking what value gave a good cutset. Setting a value of 0.8 allows the start temperature to be sufficiently high to allow a good movement of cells initially, which contributes to a good cutset.
Tstart=(psi_min-psi_max)/log(0.8)

Cooling schedule:
The temperature is allowed to cool very slowly. Every new temperature is 0.96 times the old temperature

No of cells to swap within each temperature
Within a given temperature, most of the cells are allowed to swap with each other. Our algorithm allows for swap of 80% of cells from a given side.

Challenges faced :

One problem which we overcame very early in the implementation was the choice of data structure. Initially we opted for a singly linked list. However, this proved inefficient since it took a lot of time to store the netlist. Time taken for this operation was in several minutes for the biggest benchmark. Changing the singly linked list to a doubly linked list proved to be a much better option since the time taken to store the biggest benchmark was reduced to just a few seconds. This was done by using additional data structures called net_node and cell_node to create the linked lists
Another challenge faced was with respect to the movement of pads. Since pads have 0 area, there is no restriction on their movement. Since we do not want to end up with a solution that moves all pads in one partition, after the initial partition the movement of pads is completely restricted.

The third challenge faced in the algorithm was the computation complexity. Initially the algorithm calculated the complete cutset after swapping cells. For every swap, the cutset calculation was an order of complexity = number of nets .This increased the runtime for huge benchmarks. This was overcome by calculating only the change in cutset after every swap by using a Boolean variable inside the net data structure to denote whether net is a part of data structure. With this method, the computation complexity for every swap = number of nets connected to swapped cells. For the biggest benchmarks (ibm16 and ibm18). this reduced the computation time from several hours to under 5 minutes.

IMPLEMENTATION DETAILS:

Cells and nets are created from the .are and .net files. In addition, two other data structures called net_node and cell_node are also created:

Cell:

The data structure for cell(fig1) contains the following elements:

- Cell_name - Name of cell
- Cell_area- Area of cell
- "Part" – stores the partition information of the cell
- "move" – Boolean variable that stores marks the cells as moved or unmoved
- Head – stores the address of a data structure called net_node

Net_node:

Net_node contains two elements:

- NP-Contains the address of the net
- next- contains the address of the next node in the linked list

Net:

The data structure for cell(fig3) contains the following elements:

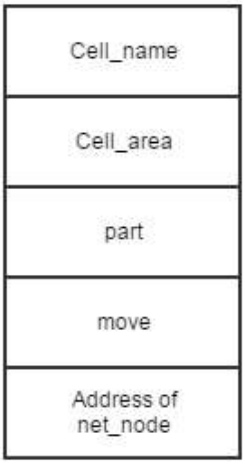
- Net_number
- Critical– Boolean variable that denotes whether the net is a part of the cutset
- Head – stores the address of a data structure called cell_node

Cell_node:

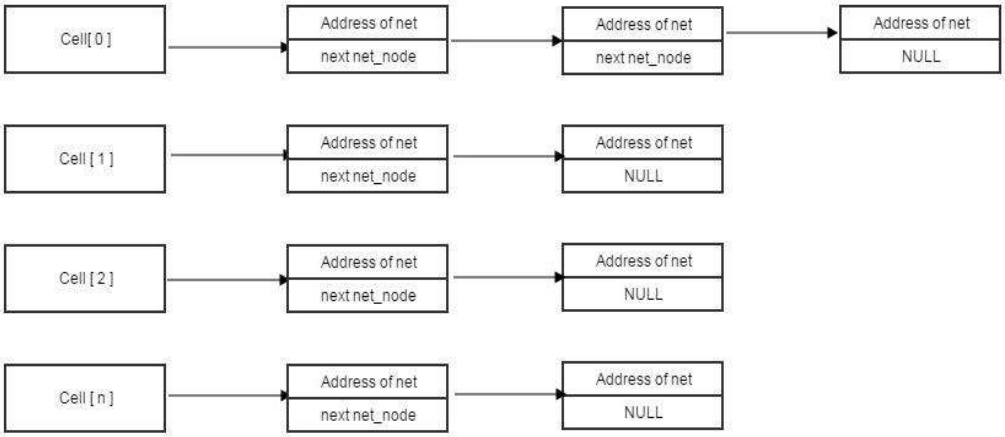
Net_node contains two elements:

- CP-Contains the address of the cell
- next- contains the address of the next node in the linked list

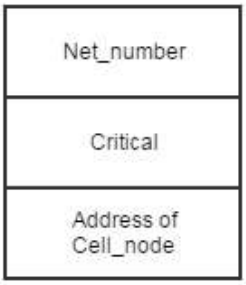
Cell_nodes and Net_nodes are used to create two linked lists which denote the connections between cells and corresponding nets (fig2 and 4). To store the array of each data structure cell and net, vectors of the standard template library (STL) container class of C++ are declared. Vectors are preferred over an array since vectors are dynamically allocated and deleted without requiring the use of new and delete operator.



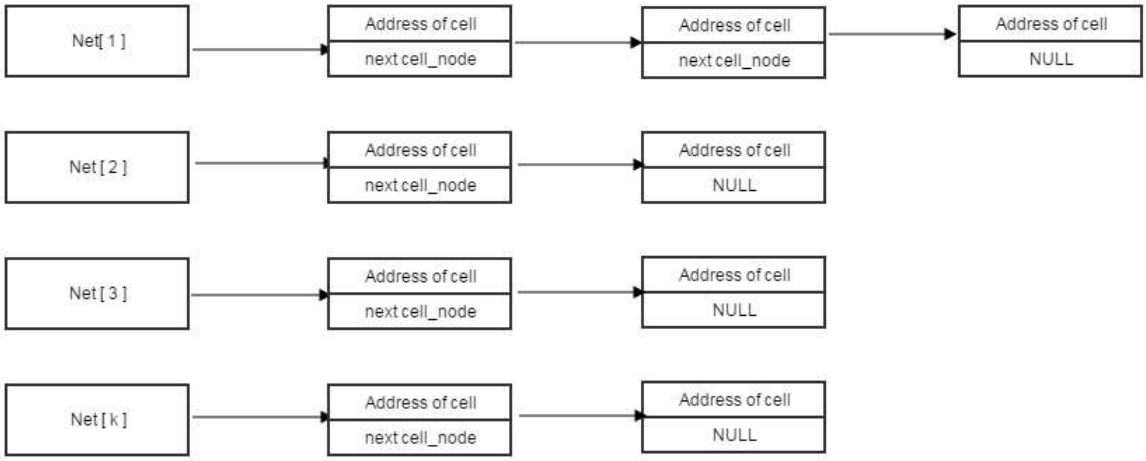
(fig1- Structure of Cell)



(Fig2 – Cell List –connection of cells to nets)\



(Fig3 – Structure of a cell)

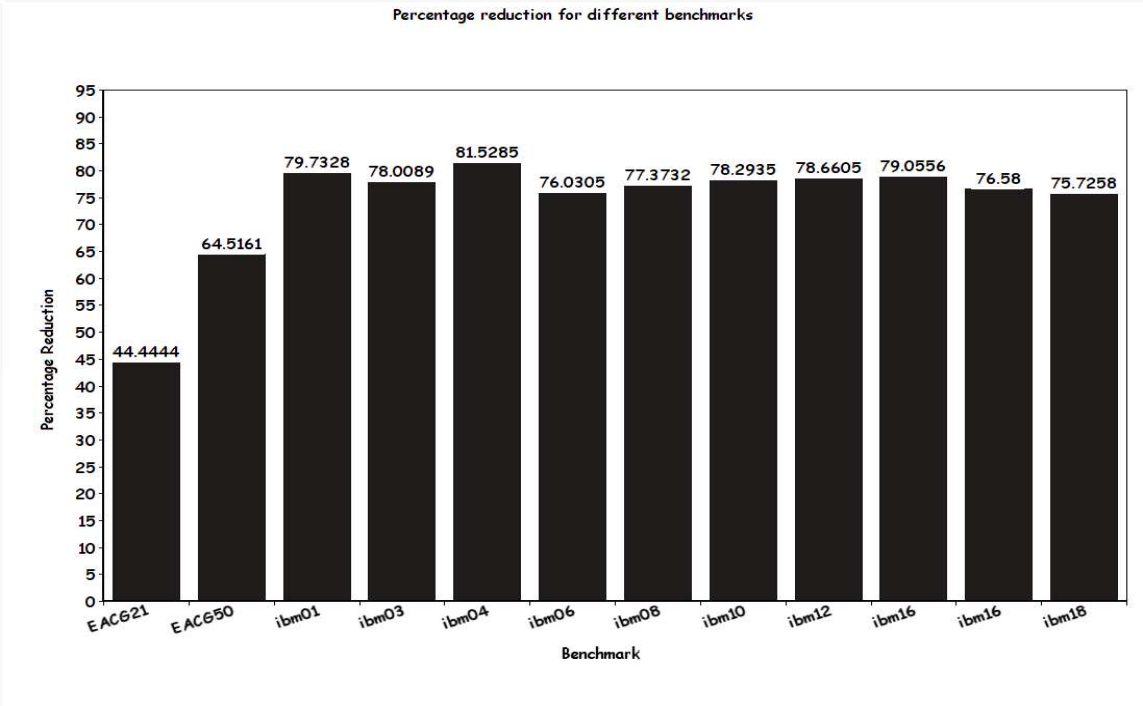


(Fig 4 – Net List – Connection of nets to cells)

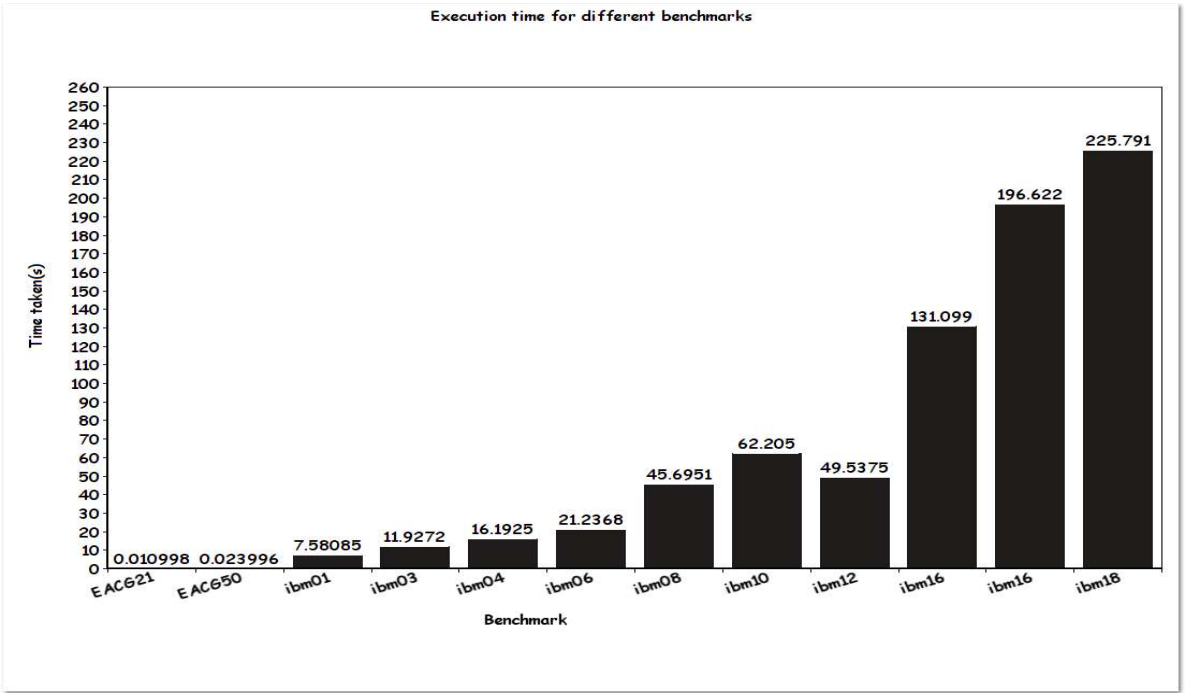
RESULTS:

The following values are obtained when all the benchmarks are tested on the UTD engnx servers. The values are recorded and plotted. We observe an average reduction in cutset by about 77% on average for all the benchmarks. The time taken for the execution of the benchmarks is observed to be reasonable. Ratio cut is calculated as *"Final cut/ ((Fractional size of partition A)*(Fractional size of partition B))"*

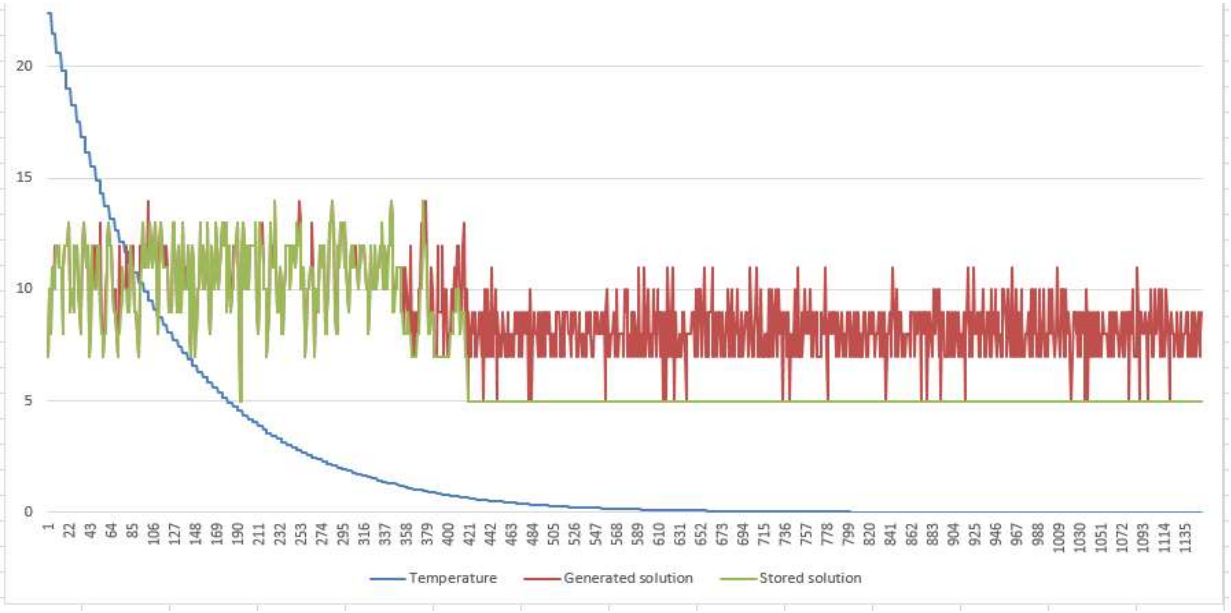
BIPARTITIONING RESULTS					
Benchmark Name	**Execution time(seconds)**	**Starting cut**	**Final Cut**	**Percentage change**	**Ratio Cut**
EACG21	0.010998	9	5	44.4444%	20.192
EACG50	0.023996	31	11	64.5161%	44.3743
ibm01	7.58085	9355	1896	79.7328%	7584
ibm03	11.9272	17357	3817	78.0089%	15335.7
ibm04	16.1925	20621	3809	81.5285%	15243.9
ibm06	21.2368	22854	5478	76.0305	22119.4
ibm08	45.6951	33531	7587	77.3732%	30607.6
ibm10	62.205	50653	10995	78.2935%	44011.2
ibm12	49.5375	49973	10664	78.6605%	43086.2
ibm14	131.099	101545	21268	79.0556%	85073.1
ibm16	196.622	130098	30464	76.5838	121935
ibm18	225.791	139127	33772	75.7258%	135187



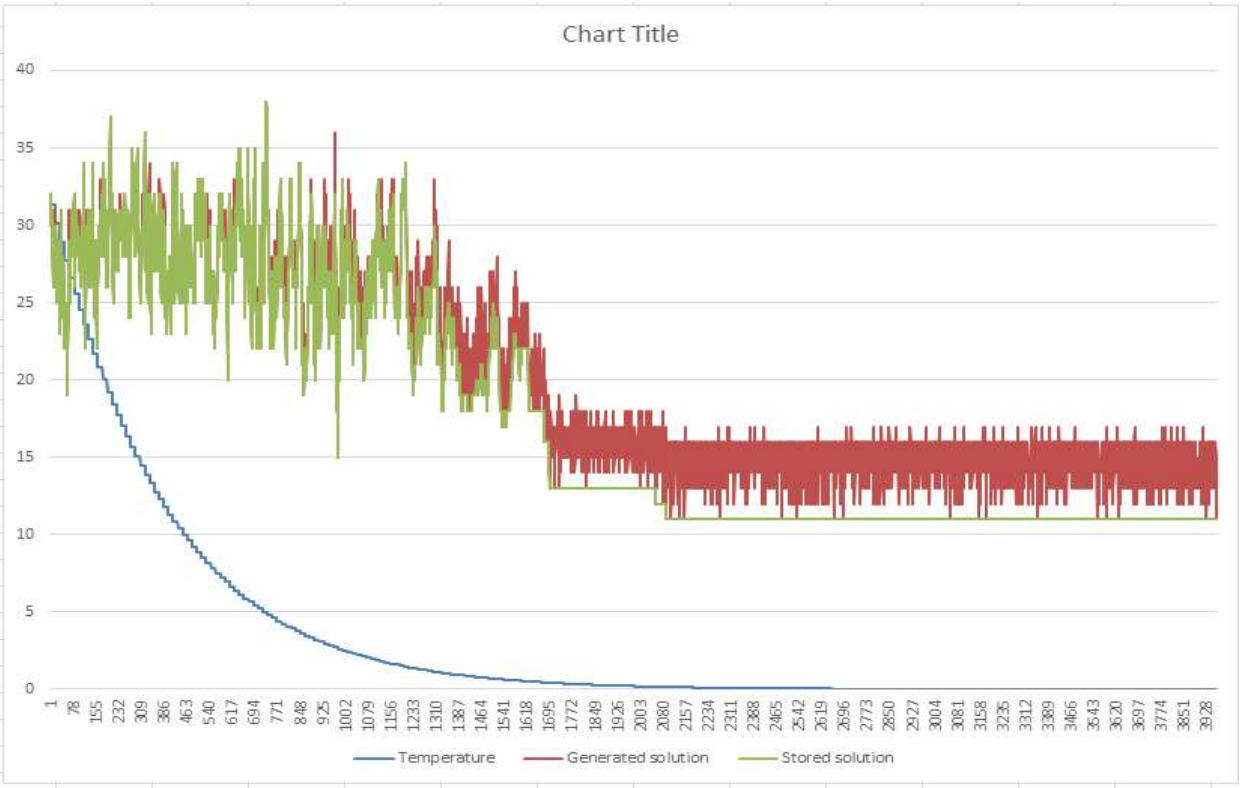
(Fig5 - Plot of Cuset reduction for various benchmarks)



(Fig 6 : Plot of execution times for various benchmarks)



(Fig7- Plot of cutset vs iteration , along with temperature for EACG21)



(Fig8- Plot of cutset vs iteration , along with temperature for EACG50)

CONCLUSIONS:

Observations from the graphs:

- Initially even very bad solutions are accepted
- As the temperature cools down, the acceptance rate decreases drastically, and only good or relatively good solutions are accepted.
- Allowing cells to initially move freely without much restriction ultimately contributes to a good cutset reduction.

We see two possible improvements over our implementation:

1. There is a possibility of reaching the best possible solution with the algorithm, and losing it due to a low acceptance probability. If we check for the value of cutset at each iteration and store it, we might compromise on the execution time. The algorithm could be implemented in such a way as to check the cutset at each iteration only after a specified low temperature, and store the lowest possible value.
2. In our implementation, the pads are stationary to prevent all pads from landing in one side of the partition. After the cells are partitioned, a separate routine could be written to swap only the pads and reduce the cutset further. In this routine, a restriction should be placed on the number of pads on each side, and not the area.

REFERENCES:

[1] Dimitris Bertsimas and John Tsitsiklis- Simulated Annealing, 1993 Vol. 8 No1, 10-15
[2] Dr. Dinesh Bhatia, Class notes and lectures of fall 2015,UT Dallas
[3] B. Stroustrup , "Programming: Principles and practice using C++ "
[4] Naveed Sherwani, ,"Algorithms for VLSI physical Design automation", Third edition.