Polish Expression based Floorplanning: Simulated Annealing

Anirudha Kurhade, *akurhade3@gatech.edu* Moumita Dey, *mdey6@gatech.edu*

I. Introduction

Floorplan design is the first step of VLSI circuit layout. The goal of floorplanning is to determine the location of the blocks (modules) so that the blocks do not overlap with each other. The objective of the placement problem is to *minimize the area* of the bounding box rectangle containing all the modules, and at the same time *minimize the total interconnection wirelength*—which in our case—is the *Half Perimeter Wire Length (HPWL)* [1].

In this paper, a heuristic space-search algorithm to find a near-optimal solution of the placement problem using Polish Expressions and Simulated Annealing is implemented. The search space of this algorithm cannot always contain an optimal solution. A slicing structure is one whose rectangular dissection can be obtained by repetitively subdividing rectangles horizontally and vertically. This representation trades off between guarantee of globally optimal solution and ease of representation. Each slicing structure is represented by a slicing tree, which is a binary tree where each internal node represents a vertical cut line or a horizontal cut line, and each leaf a basic rectangle, as shown in Fig. 1. This results in radically simple modification and realization of floorplans. To further speed-up these tasks *we offer libraries with strong and efficient Data Structures.*

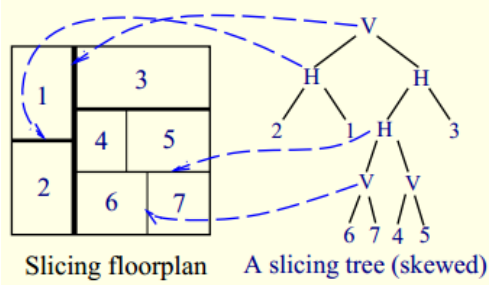


Figure 1. A slicing floorplan and its corresponding slicing tree [5].

Wong and Liu presented a method named Normalized Polish Expression [1] to represent slicing floorplans. Given a binary tree that represents a slicing floorplan of *n* blocks, the Polish Expression of this tree is a string of length 2*n* – 1 that consists of the block numbers and * for vertical cut and + for horizontal cut. The modules names are called operands and * and + are the operators. A Normalized Polish Expression should satisfy the following properties:

1. Each block appears exactly once in the string.
2. Number of operands is larger than number of operators at all points in the string – balloting property.
3. There should be no consecutive operators of the same type in the string – normality property.
4. The Normalized Polish Expression has a 1-to-1 correspondence to a slicing floorplan.

This report has been organized in the following way. We first talk about the problem formulation which includes a top level view of the algorithm implemented. The next section delves deep into the actual implementation of the algorithm, structures and libraries used, explanation of the working of the algorithm and optimizations incorporated, the extensions and modifications performed over the algorithm and a user's guide to the GUI. The floorplans obtained and statistics are discussed in the experimental results section. Finally, the main takeaways are detailed under the conclusions section of the report.

II. Problem Formulation

A. Objective

To obtain a floorplan of all the modules with an optimized cost function minimizing Area and Wire Length using Simulated Annealing and Stockmeyer Algorithm for Polish Expression and display the final floorplan using OpenGL.

B. Input

A file containing list of all modules along with their dimensions and the netlist.

C. Approach

First task is to read all the module and netlist information from the input file. An initial random Polish Expression needs to be formulated which is fed to the Simulated Annealing algorithm. This is an iterative heuristic method consisting of different moves that make changes to the structure of the floorplan. It allows the cost of implementation to go uphill in the hope of converging to a better local minima. For post processing, Stockmeyer algorithm, which focusses on minimizing the area by tabulating possible combinations of modules' rotation, is used for further optimization of the floorplan obtained after Simulated Annealing.

D. Output

An interactive GUI using OpenGL that displays the initial floorplan used and the final stages of the optimized floorplans is implemented. Various other statistics such as run time, HPWL, area reduction and chip area utilization are reported.

III. Algorithm Discussion, Implementation Issues and Extensions

The entire algorithm is implemented in C++. A coarse flow of the algorithm is illustrated in Fig. 2 (left). The entire code begins with first taking the input file path from the command line itself, enabling any file to be taken as input. The module names, widths and heights are read and provided to the data structures of the program. As our algorithm is applicable currently only to hard blocks, the pre-placed blocks provided have been considered as movable hard blocks whereas the soft blocks use a fixed aspect ratio averaging its highest and lowest permissible aspect ratios. Besides, the netlist is also read and provided to the algorithm for the calculation of HPWL for cost function. Once all the information is provided, an initial Polish Expression is created as suggested in [1] but with some changes. Instead of using a Polish Expression in the format *12*3*4*....n**, we decided to assign the operators * and + randomly as this would create a more random initial floorplan allowing more room for optimization by the Simulated Annealing process. The Simulated Annealing process converges to a local minima of cost functions by making some intelligent modifications to the floorplan. Since the starting point plays an important role in such optimizations it is fruitful to start with some random floorplan. Once the Polish Expression is obtained, the slicing tree is created and the bounding

box of each cut corresponding to each internal node is updated. The data structures are designed in such a way that we can hack into the Polish Expression paradigm and slicing tree paradigm through the same data structure. A C++ object called PE is created which is an array of all the Nodes—the modules as well as the partition nodes. These Node objects have additional fields like left, right and parent which takes care of the slicing tree representation too. In addition to that, there are several other member functions built into the object which will facilitate fast development of advanced algorithms using these data structures.

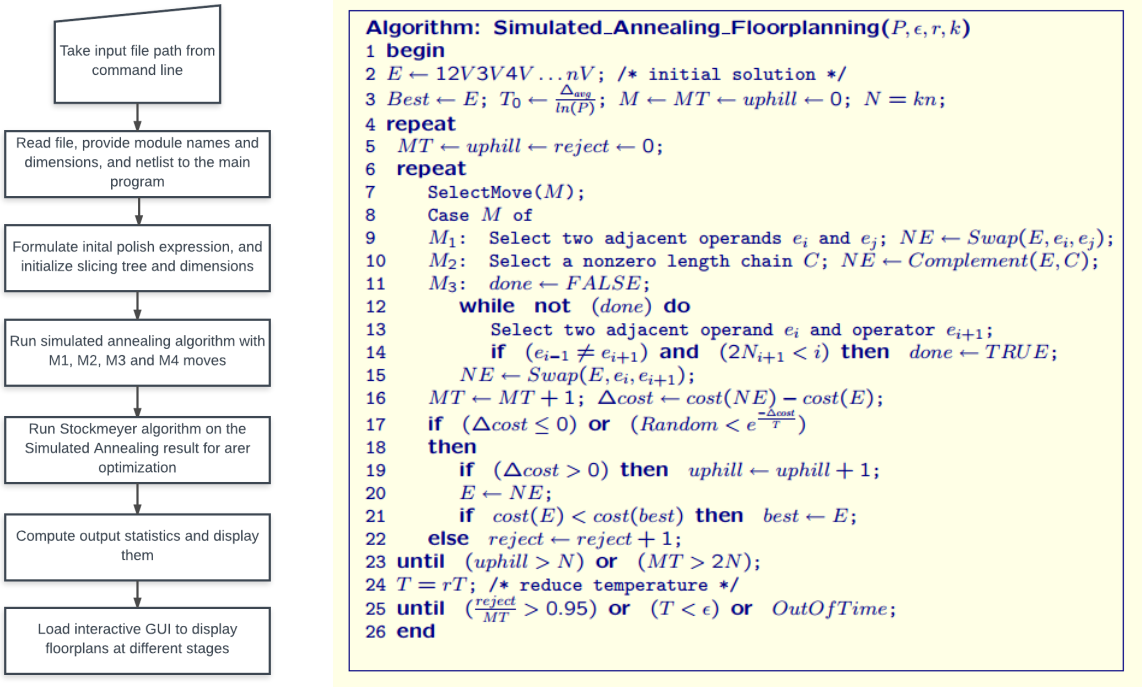


Figure 2. Overall program flow (left) and Simulated Annealing pseudo code (right) [5].

The Simulated Annealing algorithm as described in Fig. 2 (right) is now executed. The initialization of the parameters used in Simulated Annealing are described as follows. The initial temperature is set to 500 instead of what is described in the pseudo code for simplicity and this change did not have a radical change if used otherwise. The final temperature has been set to 1. The cooling rate, after some trial and error was determined to be 0.9, as for higher values, the floorplan had reached saturation and the moves conducted led to very minor changes that did not do much improvement for the floorplan, but it only led to the run time increasing dramatically. An exit condition for the algorithm weighing the number of moves at each temperature comprises of the parameters reject and MT, which keep a count of the total number of rejected moves and the total number of moves performed. The cost of the Polish Expression is a function of both the area and HPWL. It was found that both the area and HPWL were comparable in magnitude, hence the cost function was taken simply as the summation of the chip area and HPWL for the Polish Expression at that point. And this shows tangible improvement in chip area as well as HPWL upon the completion of Simulated Annealing. Once the annealing process begins, there are three types of moves: M1 is swapping *any* two operands, M2 is complementing some operator chain (a set of consecutive operators in a Polish Expression), and M3 is swapping a pair of adjacent operand and operator and additionally, we have also introduced move M4 as an extension which rotates the operand modules. The probability of this move being selected is inversely proportional to

the temperature. This is inspired from Murata et.al [2]. It entails from the fact that as we reach the local minima, making M4 moves gives us incrementally better packing since we are guaranteed that the relative position of blocks will rarely be changed. This gave us an excellent performance to the extent that Stockmeyer Algorithm had very less and—in some cases —almost no work to do. Evidently, introduction of this move gave tangible improvement in the QoR. At each instance, we randomly select one of these four moves and randomly choose respective combination of operators and/or operands upon which the move should be performed. Besides, in the case of M3, the validity of balloting and normality properties of the move needs to be determined. A modified Polish Expression is then created according to the move applied. The PE object is constructed in such a way that we have to only rearrange the pointers to the Node object, thus, reducing the memory traffic significantly. If the cost of this modified Polish Expression is less than the cost of the previous Polish Expression, the move is accepted and assigned as the current Polish Expression—on which the later modifications will be made—otherwise, the acceptance of the move is decided by a temperature based probability function. We keep a track of the best Polish Expression attained and keep this updating continuously to get the best floorplan the algorithm spanned. We have created efficient copy constructors and assignment operators to copy our entire PE object into another with—for example—as simple instructions as PE1 = PE2 (which copies PE2 into PE1 object). When the temperature is very high, there is a higher probability of a bad move being accepted, known as uphill moves, and a lower chance of acceptance at lower temperatures. To get the best out of this algorithm we have made an educated heuristic that the moves like M2 and M3 are selected more frequently at higher temperatures. After making all the moves at the current temperature level, the temperature is reduced by the cooling rate and the process is repeated iteratively until the temperature reaches the final temperature, or the number of moves accepted is sufficiently low—in which case the algorithm is consistently going uphill. We thus record the best Polish Expression obtained during the entire process and report that as our best solution.

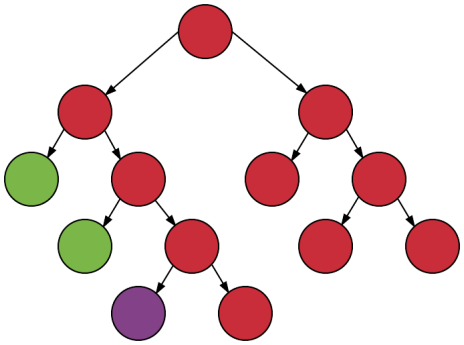


Figure 3. How update is performed. Red nodes indicate the nodes whose values will change, green nodes indicate whose values remain the same, the purple node is the node on which the move occurred.

There were many optimizations performed while coding that increases the efficiency and modularity of our code. As mentioned above, we have built an entire library for each node in the slicing tree with numerous attributes which has made the program extremely modular and can be used by anyone who wishes to implement any extensions or implement new algorithms—based on Polish Expressions—altogether. For instance, each term in the Polish Expression, be it operand or operator, can be accessed using a Node object. Using the Node object, we can determine if it is an operator or an operand, its value, its width, height, lower left x and y coordinates, the total number and the name of the nets associated with each module, pointers to its parent node and its left and right child nodes, along with

numerous functions to modify the same. Similarly, each net too can be accessed using a Net object which can access its value, its HPWL and the modules present in its net. The Net object also has the capability of computing the HPWL needed for Simulated Annealing. Also, a PE object has been provided that comprises of Node objects for accessing the modules present in its Polish Expression.

It has been observed over time, that the major factor that slows down an entire algorithm is majorly contributed by search algorithms. In order to speed up this process, hash maps was used to access the pointer to each module in O(1). After every move is performed, there are nodes that now have new values of parent, left and right children, x and y coordinates and so on. To avoid the entire tree to be updated every time a move occurs, a new update method was created. For instance, consider the example shown in Fig. 3. Among the nodes involved in a move, the purple node is the node with the lowest index in the Polish Expression array after the move is made. Only the nodes present on its left in the Polish Expression will not have any changes in their dimensions, but the ones on the right will be changed. The slicing tree corresponding to this is shown in Fig. 3. The red nodes are the ones that lie on the right of the purple node. However, the x-y coordinates of the nodes present on the left of the purple node in the Polish Expression will remain the same.

M3 is the only move that leads to a lot of structural changes. Hence it is cumbersome to keep updating the entire slicing tree every time an M3 move is made. To avoid it, a lot of observations were made and incorporated for move M3. Consider Fig. 4 (left) whose Polish Expression is 123H4VH. Consider the nodes 3 and H for move M3. The corresponding tree after the move is done is shown in Fig. 4 (right). In general, it was observed that whenever there is an operand first and operator later in the chain that needs to be swapped and the operator is the *left* child of its parent, we traverse along the parent chain of the operator till we reach to a forefather node which is *right* child of its parent. In this case we traverse to the V node. We then swap the left child of the parent (Let's call this P node) of that forefather node (V in this case) with our operator to be moved, make the left child of our operator its own right child, make the left child of the P node the left child of our operator to be moved, and move our operator's right child (which is actually the operand that involved in the move that is 3 in this case) to the position where our operator used to be. Now consider another case as shown in Fig. 5. The corresponding Polish Expression is 123H4VH and the M3 move is to be performed on 3H where H operator is now the right child. In this case, we just traverse to the next parent of the

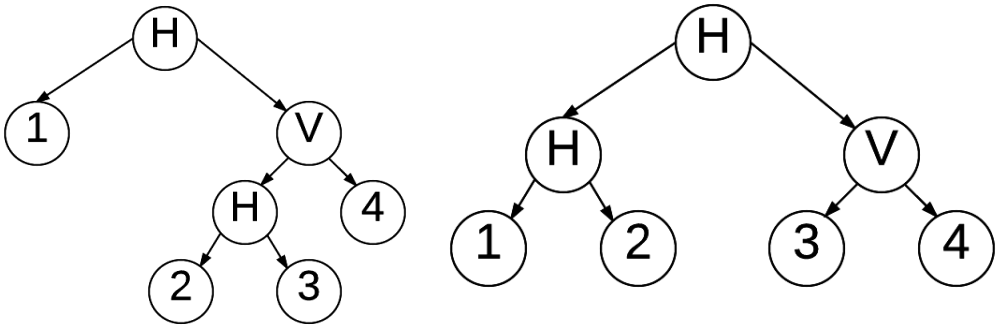


Figure 4. Illustration of M3 move when the chain to be swapped has operand first and then operator and operator is the right child.

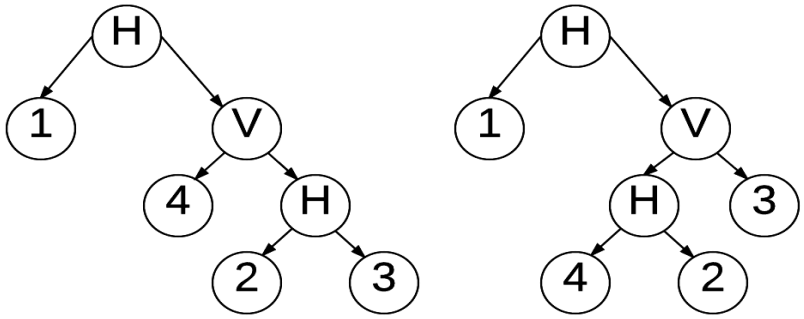


Figure 5: Illustration of M3 move when the chain to be swapped has operand first and then operator and operator is the left child.

operator, swap the left child of that parent with the operator and make it the operator's left child, shift the operator's left child as the right child, and shift the operator's right child as the operator's parent's right child. Similarly, if these same moves were to occur but with operator first and operand next, these exact same update strategies were used in the reverse order. These optimizations will considerably reduce the update time of trees every time M3 move is performed specifically when the trees are big and majority of the moves happen to be made towards the later portion of the Polish Expressions.

Once the Simulated Annealing is complete, post-processing area optimization is performed using Stockmeyer algorithm [3]. The goal is to traverse the internal nodes in the slicing tree in a bottom-up fashion so that we can compute the candidate dimensions of each internal node and its orientation. When we obtain the dimension list of the top node in the tree, we choose the one with the minimum area. We then traverse the tree in a top-down fashion to back trace the dimensions for the internal nodes as well as the orientation of the leaf nodes that contributed to the minimum dimension. [4] While implementing Stockmeyer algorithm, special attention was given while creating the data structures to enable fast back tracing, as shown in Fig. 6. We store a vector comprising of width and height of the partition nodes followed by the widths and heights of its left and right child respectively. The left and right childdimensions are not needed for a module node.

Dirty bit (0/1)	$Width_{low/high}$	$Height_{low/high}$

(a)

Dirty bit (0/1)	$Width_{computed}$	$Height_{computed}$	$Width_{left}$	$Height_{left}$	$Width_{right}$	$Height_{right}$

(b)

Figure 6. Data structures used for Stockmeyer algorithm for (a) operands (b) operators / partitions

This is followed by extracting important statistics from the execution of the algorithm such as runtime, HPWL and chip area values after every stage of the algorithm and comparing them by calculating the percentage reduction, and obtaining the chip area utilization by calculating the white space present in the floorplan. All the statistics is displayed on the terminal. At the end stage, an interactive GUI opens which was created using OpenGL graphics library in C++. This is far better than using Matlab for displaying floorplans as it is cumbersome to manually switch platforms every time you need to view the results. The integration of OpenGL in the existing code itself, made it a complete package.

The GUI is designed to display various steps of the algorithm using simple key-presses. To make the user interface seamless, a menu appears at the top right window showing the valid commands of displaying the initial floorplan, the floorplan after Simulated Annealing process and the floorplan after Stockmeyer algorithm. A grid with scale has also been incorporated to provide an estimation of the relative sizes of the modules.

IV. Experimental Results

For simplicity purpose, a sample test file (Appendix A) was made to demonstrate the various stages in the entire algorithm. Upon execution, the results obtained are illustrated in Fig. 7 and the statistics are displayed in Fig. 8. This

Figure 7. Floorplans obtained during the run for test.fp. (left) Initial floorplan, (middle) floorplan obtained after simulated annealing, (right) floorplan obtained after Stockmeyer algorithm

HPWL	
Initial HPWL	72
SA HPWL	42
Stockmeyer HPWL	42
SA HPWL reduction	41.67%
Stockmeyer HPWL reduction	41.67%
Total Chip Area	
Initial Chip Area	225
SA Chip Area	104
Stockmeyer Chip Area	96
SA Chip Area reduction	53.78%
Stockmeyer Chip Area reduction	57.33%
Area Utilization	
Total Area Occupied by modules	82
Total whitespace after SA	21.15%
Area Utilization after SA	78.15%
Total whitespace after Stockmeyer	14.58%
Area utilization after Stockmeyer	85.42%

Figure 8. Statistics for test.fp.

example demonstrates the working of both the algorithms as can be seen from the differences between the statistics obtained after simulated annealing and Stockmeyer. The simulated annealing was able to reduce the HPWL by 41.67% and there was no further reduction in HPWL due to Stockmeyer as the positions of the modules remain the same and only the orientation of modules is what changes. However there us a considerable reduction in chip area and the amount of whitespace present. After simulated annealing, Stockmeyer managed to rotate the modules that once rotated

resulted in a better floorplan and the overall chip area utilization increased from 78.15% to 85.42%. However such changes weren't so visible in the larger circuits as most of the possible rotation moves had been already performed and any change in area due to Stockmeyer did not result in any area improvement. Hence despite implementing Stockmeyer, the M4 move of simulated annealing surpassed the optimization capability of Stockmeyer.

r	modules	run time	HPWL reduction	Area reduction	chip area utilization
0.7	49	19	68.53	82.41	83.49
0.8	49	31	63.9	83.6	86.1
0.9	49	49	67.96	82.9	86.52
0.95	49	134	75.7	82.5	87.5
0.7	32	28	57.72	69.3	81.91
0.8	32	42	39.8	77	78.7
0.9	32	90	77.5	75	90.6
0.95	32	188	56	77.8	91.2
0.7	127	220	70.98	95.5	69.18
0.8	127	347	69	95.9	75.97
0.9	127	735	72.5	96.01	78.07
0.95	127	1515	79.05	95.4	67.12
0.7	18	16	65.87	75.75	83.7
0.8	18	26	72.6	76.76	83.72
0.9	18	53	66.1	71.3	75.5
0.95	18	115	77.2	74.2	80.8
0.7	95	79	71	93.8	73.1
0.8	95	121	70	93.6	70.4
0.9	95	252	75.7	93.2	67.5
0.95	95	523	76.85	93.7	73.5

Figure 9. Statistics obtained from the benchmark runs.

The statistics obtained from the different benchmarks have been tabulated as shown below. The points of concern are how the run time, HPWL reduction, area reduction and chip area utilization vary with the number of modules and cooling rate (r). The corresponding floorplans are attached in Appendix B. As the number of modules and the cooling rate increases, the run time increases drastically, thereby limiting the cooling rate to 0.9. HPWL reduction and area reduction will depend upon the initial random floorplan, hence when the initial floorplan was by itself good enough, there was lesser percentage improvement in them. Chip area utilization is a metric for amount of whitespace present and better the chip utilization, better the floorplan.

V. CONCLUSIONS

We have implemented the Simulated Annealing and the Stockmeyer algorithm which can be incorporated for floorplanning of circuits with diverse sizes with scalable time efficiency. The implementation is carried out in such a way that it will scale more efficiently in terms of runtime as the complexity and size of the circuit increases, compared to alternative approaches. The libraries are created in such a way that the future development upon this work will be extremely seamless and modular. The implementation and GUI incorporating open source C++ and OpenGL libraries makes this project a no-cost deliverable compared to a using any expensive tools like MATLAB. We intend to make this available to future developers.

REFERENCES

[1] Wong, D. F., and C. L. Liu. "Floorplan design of VLSI circuits." *Algorithmica* 4.1-4 (1989): 263-291.

[2] Murata, Hiroshi, et al. "VLSI module placement based on rectangle-packing by the sequence-pair." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.12 (1996): 1518-1524.

[3] Stockmeyer, Larry. "Optimal orientations of cells in slicing floorplan designs." *Information and Control* 57.2-3 (1983): 91-101.

[4] Lim, Sung Kyu. *Practical problems in VLSI physical design automation*. Springer Science & Business Media, 2008.

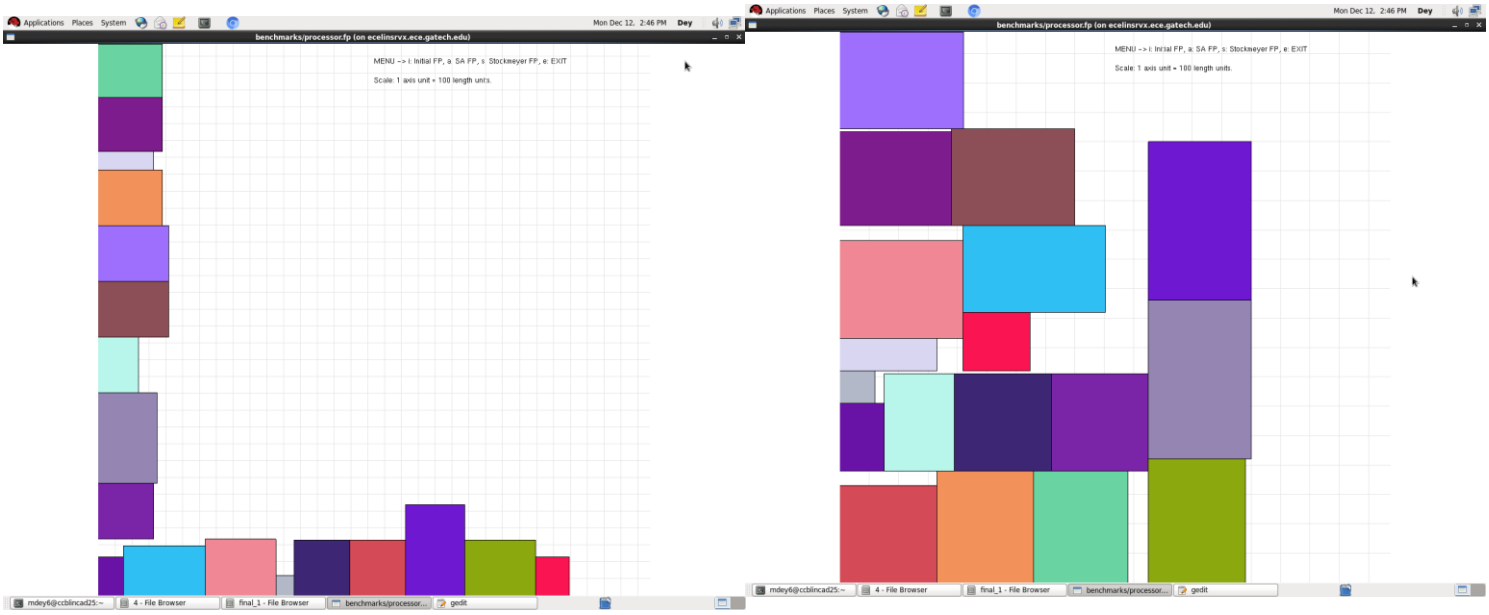
[5] Lim, Sung Kyu. *Floorplanning ECE6133 Physical Design Automation.* Georgia Institute of Technology, < http://users.ece.gatech.edu/limsk/course/ece6133/slides/floorplanning.pdf>, last accessed on December 12, 2016.

Appendix A: test.fp

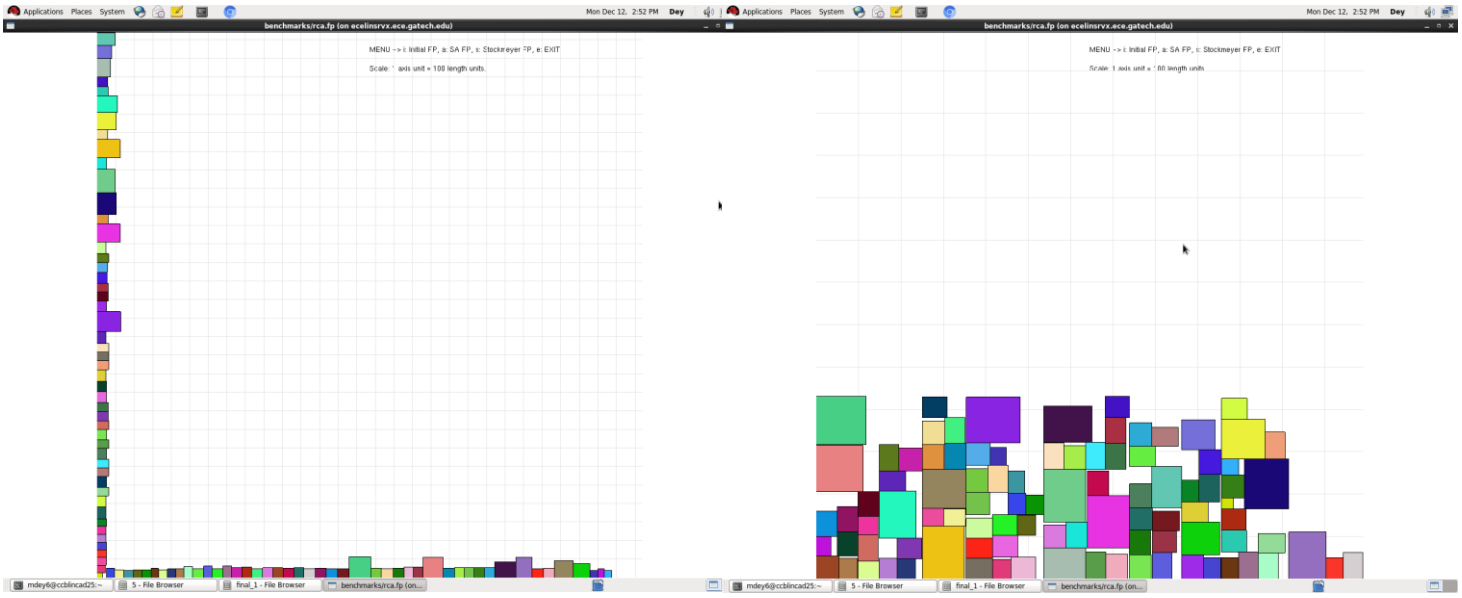
```
*PREPLACED
*END


*SOFTBLOCKS
*END



*HARDBLOCKS
m1 1 3
m2 3 2
m3 2 4
m4 3 3
m5 1 5
m6 3 5
m7 5 4
m0 2 8
*END


*TERMINALS
*END


*NETS
- n1 2
m1
m3
- n2 2
m2
m3
- n3 2
m3
m5
- n4 2
m4
m5
- n5 2
m5
m6
- n6 2
m6
m0
- n7 2
m7
m0
*END
```

Appendix B: Outputs from benchmarks (left) intial floorplan, (right) final floorplan

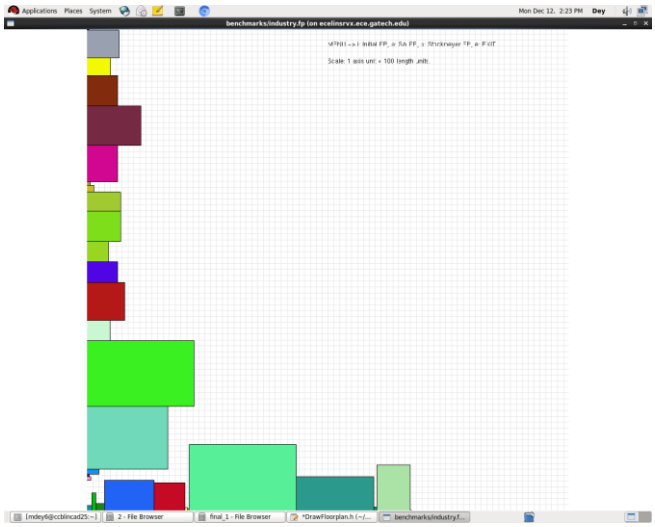
Processor.fp

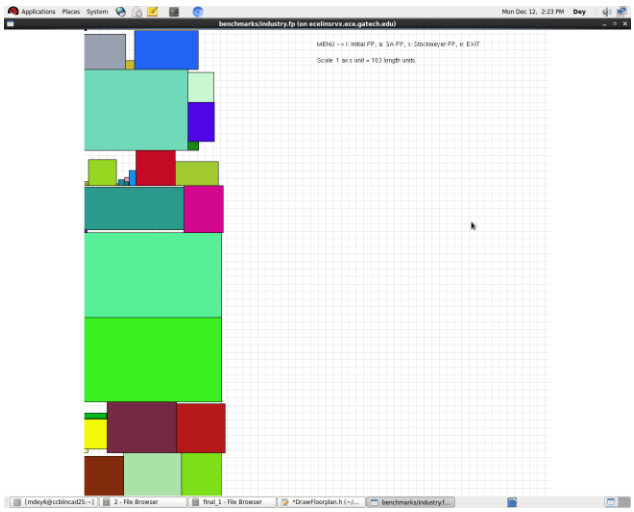


Rca.fp

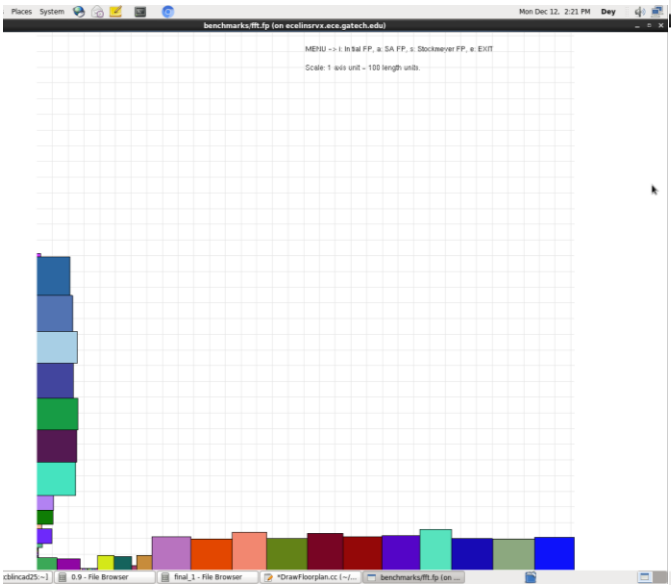


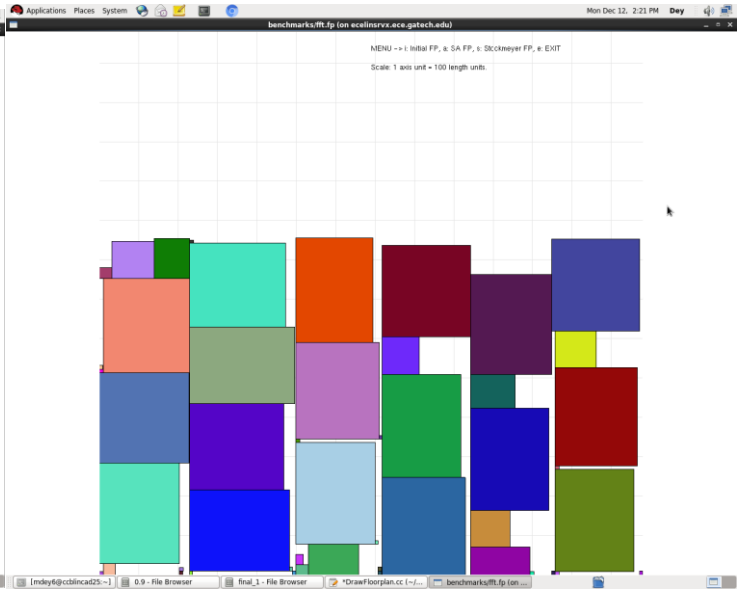
Industry.fp

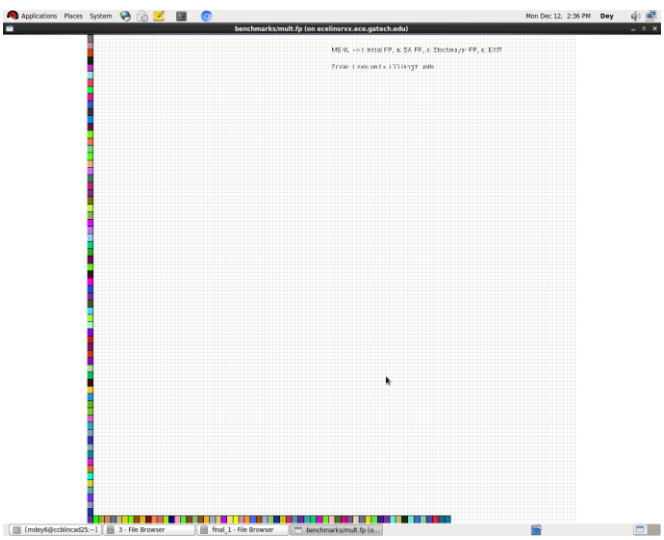




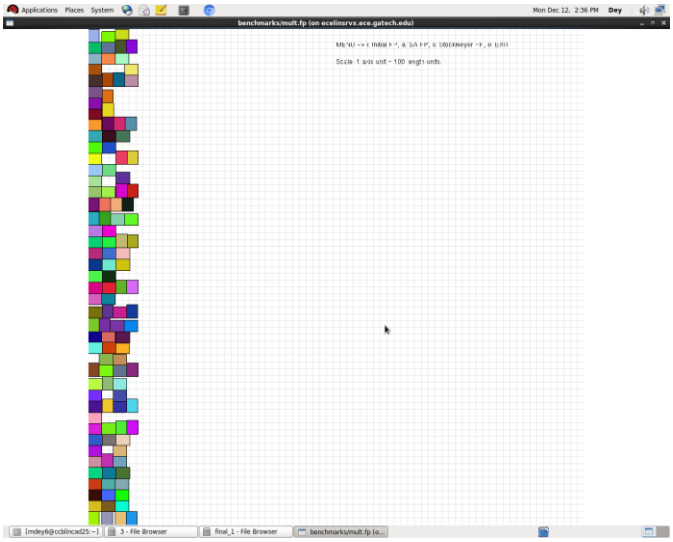
Fft.fp






Applications Places System
Mon Dec 12, 2:36 PM Dey
benchmarks/mult.fp (on ecelinsrvx.ece.gatech.edu)
3 - File Browser
final_1 - File Browser

mult.fp


Applications Places System
Mon Dec 12, 2:36 PM Dey
benchmarks/mult.fp (on ecelinsrvx.ece.gatech.edu)
3 - File Browser
final_1 - File Browser