Improved Global Routing By Using A Star Algorithm

Abdulrahman Khalid[*], Hossam Ahmed[†], Mahmoud Mohamed[‡] and Muhanad Atef[§]

Computer Engineering Dept., Faculty of Engineering, Cairo University

Cairo, Egypt

[*]abdulrahman.elshafei98@gmail.com, [†]hossamahmed201515@gmail.com, [‡]mmmacmp@gmail.com, [§]muhanad.atef23@gmail.com

Abstract—In this paper VLSI routing is improved by improving global routing, this can be done by using A-Star with a heuristic cost function that has parameters which affect the time taken by the router on changing instead of Dijkstra's algorithm in finding path, which will reduce the time taken in this process and achieve the minimum wire length, many comparisons are taken in this paper with different algorithms to find the optimum algorithm to be used to achieve both minimum wire length and minimum time taken. From the comparisons of the paper, we can find that using any algorithm is a trade-off as when the taken time is decreased, the wire length is increased and vice versa, so there is no algorithm which is better from the other algorithms in general but using A-Star algorithm with the heuristic function in finding the path is a good approach to be used in global routing as it decreases the routing time and achieves the minimum wire length.

Index Terms—VLSI Routing, Global Routing, Routing Algorithms, Fast Global Routing, Fast Routing Algorithms, Routing Algoritms Comparisons.

I. INTRODUCTION

Routing is a critical step in the physical design process. Until now the optimum solution for VLSI routing has not been achieved yet, so it is considered a very interesting challenging field. It is exactly done in two steps, global routing, and detailed routing, in global routing, A-technique for 3D global routing is to compress a *3D* grid into a *2D* grid and handle *2D* global routing. The obtained solution is then projected back to *3D* by assignment of layers. as introduced in most of modern designs as in [1], [2], [3], [4], [5], [6], [7], and [8]. At first global routing is run which is responsible for making an approximate routing for the whole circuit in order to be used as a guide for detailed routing, then the detailed routing is run to make the exact routing for the system. That means if global or detailed routing is improved the whole routing process is improved, but there are many problems that have to be overcome to make a correct routing process. First of all, the scale has to be taken into consideration, as millions of wires exist in a small chip area which means that many kilometers of wires are placed in a very small area, so total wire length has to be minimized as much as possible, also it is known that as the wire length increases the resistance increases as well which means more delay in the chip. There is another problem as circuits are made in nano-scale which means that its geometric will be complex. Another problem is that the routing algorithm has to be applicable for more than

(a) Before routing (b) Iteration number x
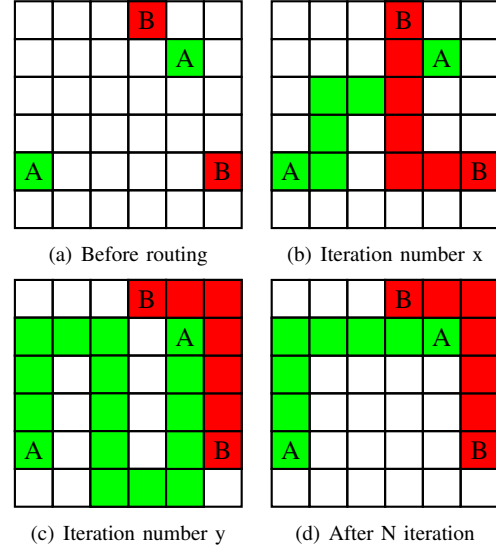
(c) Iteration number y (d) After N iteration

Fig. 1. Finding path in global routing

one layer with different costs. The direction of wires also has to be taken into consideration as the direction of wires in every layer can be either vertical or horizontal and no diagonal paths, then to go from source 'S' to target 'T' the path taken should be in (vertical — horizontal) directions that specified by the layer (at each layer wires are placed in one direction only), then there is another problem as when a wire goes from layer to another to continue on the perpendicular direction it has to go through via which has a high resistance. DFM (design for manufacturer) rules also have to be achieved. All of these constraints must be taken into consideration with the global routing to achieve a hundred percent of the circuit connections, which means global routing will take a lot of time to achieve all these constraints, and here is the challenge to achieve all the routing specifications with the minimum time taken.

Figure 2 shows a very simple approach of how the global router works. At (a) the source and target of both (A,B) need to be connected ignoring obstacles, nevertheless we can observe that the global router have to iterate to get the best routing paths, at (b) (B,B) connected but there is no way to connect (A,A) as when (B,B) connected together they blocked the way for (A,A) to be connected, after some iterations we can find

the figure at (c) in which (A,A) and (B,B) connected correctly, but there is a problem, the (A,A) connection is not the optimal path as there are paths which achieve less wire length, so the global router have to iterate until reaching the optimal path. These iterations are done for only two connections in one layer without obstacles, then how about millions of wires in VLSI? this shows how much the global routing algorithm has to be very fast in order to connect this huge number of wires as fast as possible.

II. RELATED WORK

Several papers proposed various types of approaches to improve the routing process, each of them tried to improve the overall routing by improving one or more parameters, some papers tried to decrease the number of vias, other papers tried to decrease the time taken and so on.

In [1] a sequential global routing is used and two bounded length maze algorithms as finding path algorithms are provided in order to make the router faster and to avoid congestions thus avoiding overflow, the first one is optimal-BLMR and the second one is heuristic-BLMR. optimal-BLMR is used to get the minimum cost paths to be used as routing paths, this can be done in three steps. First BLC (bounded length constraint) is defined as a greater number than Manhattan distance then to go from source to target the neighbor points are tested if it can be a part of a path or not, each point that violates the BLC constraint is discarded. Second, if the route started from point *v*, ended at point *u* and there were many paths between these two points, the normal maze algorithm will take the path with the minimum cost which may cause the route to pass through congestions, but in optimal-BLMR, it keeps track of all paths between these two points in order to choose the path that will not cause overflow, it iterates on the minimum cost path every time and if it found a suitable path it reserves that path, otherwise, it discards that path. Third step the optimal-BLMR iterates on the reserved paths and choose the one to be used for routing. Heuristic-BLMR is used to speed the router up by reserving only one path between the two points, but it has to keep the advantage of optimal-BLMR (avoiding congestions), this can be achieved by reserving the selected path only if the wire length is enough to detour around congested regions. The advantages of this paper are using sequential global routing which is based on multithreaded global routing which speedup the router between 2.71 and 3.12 in overflow free cases, avoiding collision by using optimal-BLMR, and making a fast and nearly avoiding collision algorithm (heuristic-BLMR). But there are some disadvantages too, as optimal-BLMR is very slow to be used, although heuristic-BLMR is faster than optimal-BLMR its results are not accurate as the wire length is not the best compared with other papers, and it is done on *2D* grid then it is projected to the *3D* one however, this approach gives a good result but it is not accurate like routers that apply routing in the *3D* grid directly as in [9].

In [4], both of via count and runtime are reduced, this done by integrating [10], [11], and [12] with via aware Steiner tree generation, 3-bend routing, and layer assignment with careful edge and net ordering to create [4]. Via aware Steiner tree is used to at the beginning of the global routing, it generates a suitable topology, by changing tree topology the via count greatly changed, which means that using a suitable topology for Steiner tree will greatly reduce the via count. The 3-bend routing is used instead of L, U, Z, maze, and monotonic routing, as L, U, Z routing can not avoid congestions but they generate a little number of vias, maze and monotonic can avoid congestions but they generate a lot of vias and their runtime is very high, so the 3-bend routing was used as it is fast, its completely *O(nm)*, and it generates vias less than maze and monotonic routing as it consists of two L routing. Layer assignment with careful ordering algorithm is used as a solution of *3D*, it is like all the modern techniques project the *3D* grid into *2D* one to be easier and faster in routing, this algorithm guarantees the wire length and overflow unchanged on assigning to layers, dynamic programming is used in layer assignment to make it faster. From the previous explanation, the advantages of this paper are decreasing the number of vias, which means less power consumption and circuit delay, and decreasing runtime of global routing thus decreasing runtime of whole the routing process. But it has disadvantages too, one of them is the wire length is not optimal, as using 3-bend routing increases the wire length to avoid congestions, another one is caused by layer assignment as if the *2D* was not congestion-free the results will not be accurate.

III. PROPOSED APPROACH

For global routing, three parameters are considered. The first is the total wire length, the second is the total overflow. The third is the running time. In our proposed solution, we are focused on the time parameter without affecting much the total wire length and total overflow. So according to minimize the time we have decided to apply A* algorithm as the pathfinder algorithm with a proposed heuristic function, A* is basically a guided variation of Dijkstra. This algorithm can be turned into a better or worse pathfinding algorithm by experimenting with the heuristics it uses and how it evaluates each node a* expands on a node only if it seems promising. Its only focus is to reach the goal node as quickly as possible from the current node, not to try and reach every other node. Illustrative example:
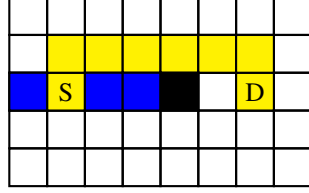
In the illustrative example the visited nodes are marked in blue color and the path is in yellow color. Both have path of six cells which in our case they have optimal wire length but totally different running time as Dijkstra visits much more nodes than A* , in the case of many target nodes, Dijkstra might be better in total wire length but much slower than A*, with an appropriate heuristic function we can minimize the time and try to reach optimal wire length. The heuristic function is a way of informing the search about the path to a target that provides an intelligent way of guessing which neighbor of the node is going to lead to the target faster. Hence, our heuristic function composed of Manhattan distance, layer index, and some constants.
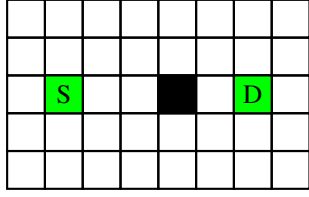
(a) Maze
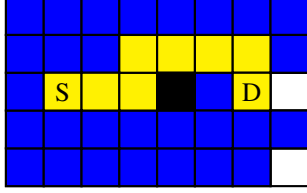
(b) Applying Dijkstra (c) Applying A*

Fig. 2. Comparison between Dijkstra & A*

$$h = \ln((\mid p1 \cdot x - p2 \cdot x \mid + \mid p1 \cdot y - p2 \cdot y \mid) * const1)$$
$$+ (\mid p1 \cdot layerid - p2 \cdot layerid \mid) * const2$$

The global router is implemented using C++ with the boost library, and a parser for LEF/DEF formats. We conducted our experiment with the same configuration and tried to test the global router behavior using A* algorithm instead of Dijkstra's.

A. Proposed Algorithm

Algorithm 1 A* Algorithm

Input: Grid values g, Source s, Destination d
Output: Path

1: **procedure** A*
    Declaration and Initialisation:
2:     Cells θ
3:     Visited Nodes ϕ
4:     Openlist Nodes δ
5:     The cost from starting point to a given node g
6:     The heuristic function h
7:     (The cost of g + The cost of h) is f
8:     **if** s or d is invalid **then return**
9:     **end if**
10:     **if** s or d is blocked **then return**
11:     **end if**
12:     **if** $s = d$ **then return**
13:     **end if**
14:     Initialize δ
15:     Initialize ϕ
16:     **while** δ is not empty **do**
17:         $currNode$ = node with least f in δ
18:         δ.Remove($currNode$)
19:         Generate 4 neighbours of $currNode$
20:         **for** $neighbour \in neighbours$ **do**
21:             **if** $neighbour = d$ **then**
22:                 Get path **return**
23:             **else if** $neighbour \notin \phi$ and $neighbour$ is not blocked **then**
24:                 $gnew = currNode.g$ + distance between neighbour and currNode
25:                 $hnew$ = distance from d to $neighbour$
26:                 $fnew = gnew + hnew$
27:                 **if** $fnew < neighbour.f$ **then**
28:                     $neighbour.g = currNode.g$ + distance between neighbour and currNode
29:                     $neighbour.h$ = distance from d to $neighbour$
30:                     $neighbour.f = neighbour.g + neighbour.h$
31:                     δ.Add($neighbour$)
32:                 **end if**
33:             **end if**
34:         **end for**
35:         ϕ.Add($currNode$)
36:     **end while**
37: **end procedure**

IV. EXPERIMENTAL ANALYSIS

A. Assumptions

The proposed scheduling algorithm is software simulated using a Python script which simulates scheduling independent CPU-bound processes on a single processor environment

which guarantees that no more than a single process is getting scheduled at any arbitrary moment. Each process is assumed to have its own predetermined burst time, arrival time and the queue to which each one belongs. The proposed approach is non-preemptive. For the sake of giving an example, if a process was lately introduced to a queue denoted by Q_i prior to the current queue getting scheduled, it won't get scheduled until the current queue, its subsequent queues and the queues prior to Q_i get scheduled.

B. Experimental Scheme

On one hand, the input arguments to the proposed algorithm implementation are the number of processes to be scheduled, their burst time, their arrival time and the queue where each one belongs. On the other hand, output parameters are the average waiting time, average turnaround time and throughput. The following equations are used to calculate the previously mentioned output parameters:

$$Average\ Waiting\ Time = \frac{Total\ Waiting\ Time}{Number\ of\ Processes} \quad (1)$$

$$Average\ Turnaround\ Time = \frac{Total\ Turnaround\ Time}{Number\ of\ Processes} \quad (2)$$

$$Throughput = \frac{Number\ of\ Executed\ Processes}{Total\ Execution\ Time} \quad (3)$$

C. Performance Metrics

As a means to have a concrete, viable evaluation of either the proposed algorithm or any other scheduling algorithm, the output parameters are taken into consideration for analysis. Since the average waiting time indicates the average time that a process had to starve for, therefore the lower the average waiting time is the better. The same principle applies to the average turnaround time and the number of context switches, as the former implies the average time spent by the process since its arrival time to its completion and the latter costs time as the CPU is assigned back and forth between different processes. Contrarily to the prior metrics, the larger the throughput is the better as it indicates the number of processes that are completely executed per unit time.

D. Simulation

For the sake of showcasing the proposed algorithm, a number of processes, their predetermined burst time values and their arrival time values are taken as input to the Python simulation script. Suppose that the input to the script is according to the following table:

TABLE I

Process	Arrival Time	Burst Time	Queue
1	0	60	1
2	0	50	1
3	0	40	2
4	0	30	2
5	0	10	3
6	0	210	3
7	0	200	3

According to the proposed algorithm, the time quanta calculated are as follows:

TABLE II

Queue	Quantum Value
1	55
2	40
3	615
4	0
5	0

All processes are sorted in ascending order according to their remaining time and are scheduled by assigning the time quantum calculated for their respective queue. The time spent scheduling a particular queue is the waiting time for its subsequent queues.
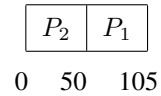
The scheduling process goes as follows:

P_2	P_1

0   50   105

Fig. 3. Q_1 Gantt Chart

Considering that each process in Q_1 is assigned a quantum value of 50, as we reach the last process in Q_1, the total time elapsed equals 105, which happens to be the time that all the other processes in the subsequent queues had to wait for, hence the addition of their waiting time by a value of 105 units of time.

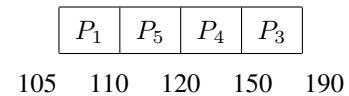| P_1	P_5	P_4	P_3

105   110   120   150   190

Fig. 4. Q_2 Gantt Chart

Even though P_5 is initially assigned to Q_3 as in Table I, it was promoted to Q_2 due to satisfying inequality (**??**).
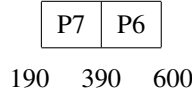
P7	P6

190   390   600

Fig. 5. Q_3 Gantt Chart

Whenever processes reach the lowest queue precompletion, they are scheduled using the RR scheduling algorithm with a relatively large quantum time value which is in most cases similar to using the FCFS algorithm because, as the time quantum value of an RR algorithm tends to infinity which could practically be a very large number relative to the available processes remaining time values, the algorithm tends to morph into the FCFS algorithm. This procedure is iterated until all the processes are finished. Simulation results are shown in the table below:

TABLE III

Avg. Turnaround Time	Avg. Waiting Time	Throughput
230	144.3	0.011667

E. Performance Comparisons

To assess the performance of the proposed algorithm implementation, multiple test cases are addressed and analyzed in seven different experiments. In each experiment, the output of the proposed algorithm implementation is compared to the output of another scheduling algorithm implementation addressed in a different paper, such as standard MLFQ algorithm with static quantum RR and other variants of MLFQ algorithms and RR algorithms.

1) Experiment 1: In this experiment, the proposed algorithm is compared against two MLFQ algorithm variants stated in b4. The first uses a static version of the RR algorithm for scheduling each queue, while the second variant uses a dynamic version of the RR algorithm for doing so.

TABLE IV
EXPERIMENT 1 INPUT

Process	Arrival Time	Burst Time	Queue
1	1	25	1
2	5	70	1
3	6	84	1
4	7	17	1
5	8	35	1

TABLE V
EXPERIMENT 1 RESULTS

Algorithm	Avg. Turnaround Time	Avg. Waiting Time
Proposed Algorithm	115	68.8
Dynamic RR MLFQ	150.8	107.6
Static RR MLFQ	161.4	116.2

2) Experiment 2: In this experiment, the proposed algorithm is compared against two MLFQ algorithm variants stated in b4. The first uses a static version of the SJFRR algorithm for scheduling each queue, while the second variant uses a dynamic version of the SJFRR algorithm for doing so.

TABLE VI
EXPERIMENT 2 INPUT

Process	Arrival Time	Burst Time	Queue
1	1	25	1
2	5	70	1
3	6	84	1
4	7	17	1
5	8	35	1

TABLE VII
EXPERIMENT 2 RESULTS

Algorithm	Avg. Turnaround Time	Avg. Waiting Time
Proposed Algorithm	115	68.8
Dyn. SJFRR MLFQ	134	91.8
Stat. SJFRRMLFQ	143.4	98.2

3) Experiment 3: In this experiment, the proposed algorithm is compared against two MLFQ algorithm variants stated in b5. The first uses a static version of the SJFRR algorithm for scheduling each queue, while the second variant uses a dynamic version of the SJFRR algorithm for doing so.

TABLE VIII
EXPERIMENT 3 INPUT

Process	Arrival Time	Burst Time	Queue
1	0	8	1
2	3	133	3
3	2	21	2
4	8	39	2
5	19	67	2
6	33	114	3
7	33	54	2

TABLE IX
EXPERIMENT 3 RESULTS

Algorithm	Avg. Turnaround Time	Avg. Waiting Time
Proposed Algorithm	151	88.7
Dyn. SJFRR MLFQ	252	119
Stat. SJFRR MLFQ	351	228

4) Experiment 4: In this experiment, the proposed algorithm is compared against multiple variants of the MLFQ

algorithm that are stated in b6: standard MLFQ algorithm, a priority-based MLFQ algorithm and a vague logic-based MLFQ algorithm.

TABLE X
EXPERIMENT 4 INPUT

Process	Arrival Time	Burst Time	Queue
1	0	40	1
2	0	30	1
3	0	50	1
4	2	70	1
5	4	25	1
6	6	60	1
7	7	45	1

TABLE XI
EXPERIMENT 4 RESULTS

Algorithm	Avg. Turnaround Time	Avg. Waiting Time
Proposed Algorithm	185.85	140.14
VMLFQ	190	170
MLFQ	232.14	175
PMLFQ	240	180

5) Experiment 5: This experiment is the same as the previous one, but with a different input test case.

TABLE XII
EXPERIMENT 5 INPUT

Process	Arrival Time	Burst Time	Queue
1	0	90	1
2	0	30	1
3	0	28	1
4	0	57	1
5	0	73	1
6	0	19	1
7	0	42	1
8	0	67	1

TABLE XIII
EXPERIMENT 5 RESULTS

Algorithm	Avg. Turnaround Time	Avg. Waiting Time
Proposed Algorithm	212.5	161.75
VMLFQ	260	225
MLFQ	290	240
PMLFQ	300	245

6) Experiment 6: In this experiment, the proposed algorithm is compared against two variants of the RR algorithm stated in b3. The first is a static version of the RR algorithm with a constant quantum value of 25 for scheduling each queue while the second uses a dynamic version of the RR algorithm called SRBRR for doing so.

TABLE XIV
EXPERIMENT 6 INPUT

Process	Arrival Time	Burst Time	Queue
1	0	13	1
2	0	35	1
3	0	46	1
4	0	63	1
5	0	97	1

TABLE XV
EXPERIMENT 6 RESULTS

Algorithm	Avg. Turnaround Time	Avg. Waiting Time
Proposed Algorithm	113.2	62.4
Dynamic SRBRR	122.4	71.6
Static RR	148.2	97.4

7) Experiment 7: This experiment is the same as the previous one, but with a different input test case. Note that for this test case, a process queue number is irrelevant to both the RR algorithm and the SRBRR algorithm mentioned in b3.

TABLE XVI
EXPERIMENT 7 INPUT

Process	Arrival Time	Burst Time	Queue
1	0	54	1
2	0	99	3
3	0	5	2
4	0	27	2
5	0	32	2

TABLE XVII
EXPERIMENT 7 RESULTS

Algorithm	Avg. Turnaround Time	Avg. Waiting Time
Dynamic SRBRR	93.6	50.2
Proposed Algorithm	106.8	63.4
Static RR	152.2	108.8

F. Observation

From the above simulations of different test cases and multiple performance comparisons that involved as many as 11 different scheduling algorithms not including this paper

algorithm, it is clear that the average turnaround time and the average waiting time of the proposed algorithm is less than or – in few occasions – nearly equal to those of the stated algorithms. With that said, the proposed algorithm is arguably advantageous over those algorithms, considering even the case in which it underperformed compared to the SRBRR algorithm, it is still favourable due to the capability to separate processes into categories based on their need for the processor and other advantages of the MLFQ algorithm. The performance of the proposed algorithm compared to other algorithms is further illustrated in the following graphs:
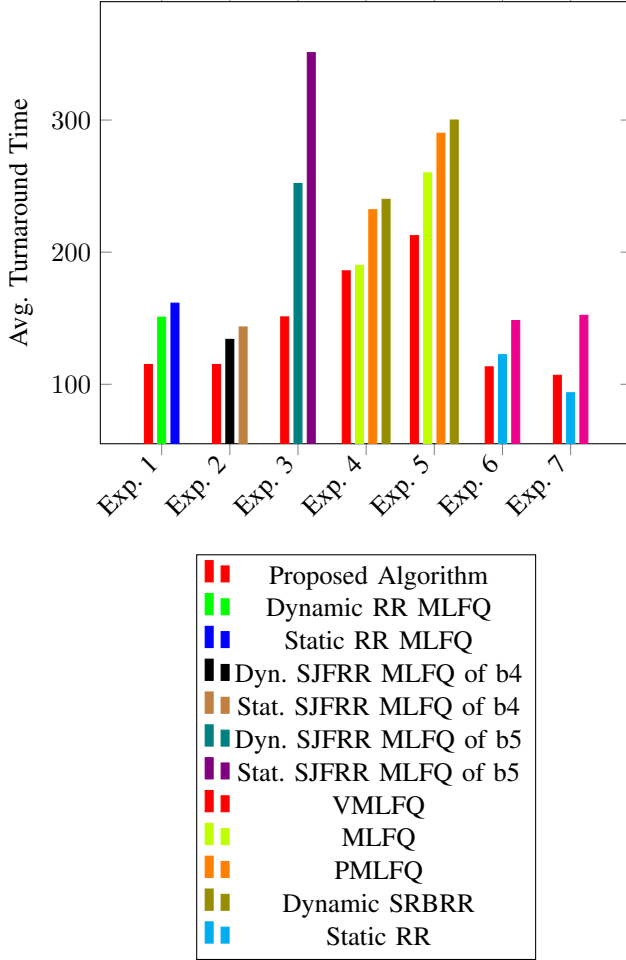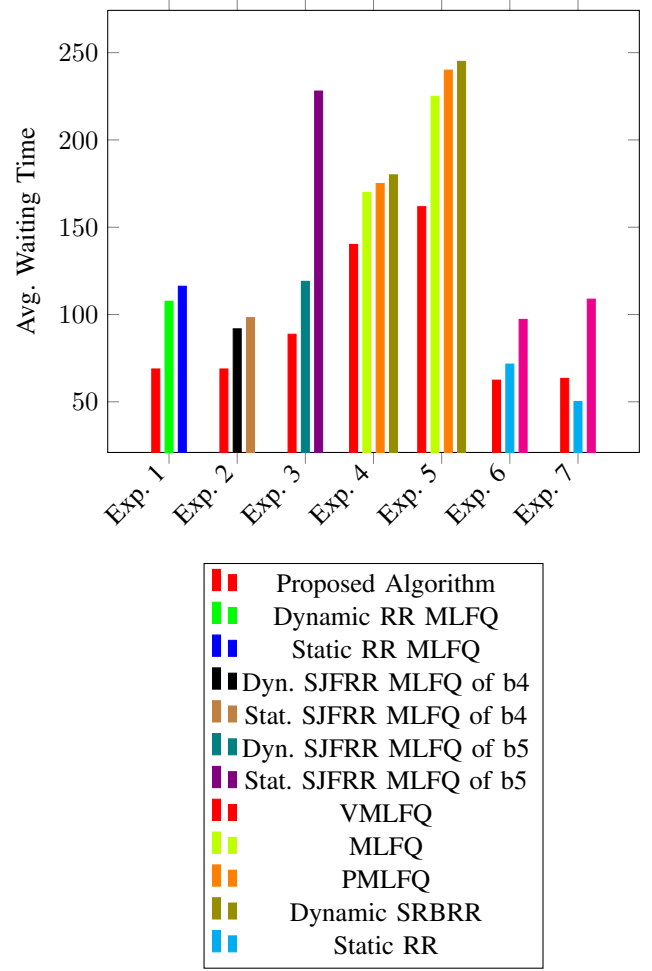

Fig. 7. Comparison graph for average waiting time

V. Conclusion and Future Work

The goal of this paper is to tackle different shortcomings associated with the standard MLFQ scheduling algorithm as well as its variants discussed in several papers. To resolve these deficiencies, we introduced different adjustable policies and techniques. It is evidently clear that those methods yield better CPU performance and optimize utilization by reducing the average waiting time as well as the average turnaround time. Despite experimenting numerous combinations of the parameters and scheduling policies by which the proposed MLFQ algorithm operates, we can say that there is yet a large room for experiment and improvement through finding better methods and policies which would make the algorithm more adaptable to the nature of the submitted processes and overall more enhanced. For instance, adjusting the criteria by which the schedular decides whether to promote a process might further lessen starvation. Making the proposed algorithm preemptive might also aid mitigating starvation and render overall better performance. The possibilities are limitless.

References

[1] Muhammet Mustafa Ozdal and M. D. F. Wong, "Archer: a history-driven global routi algorithm," *2007 IEEE/ACM International Confer-*

Fig. 6. Comparison graph for average turnaround time

ence on Computer-Aided Design, San Jose, CA, 2007, pp. 488-495, doi: 10.1109/ICCAD.2007.4397312.

[2] Chang, Yen-Jung, Yu-Ting Lee, and Ting-Chi Wang. "NTHU-Route 2.0: a fast and stable global router." *2008 IEEE/ACM International Conference on Computer-Aided Design. IEEE*, 2008.

[3] H.-Y. Chen, C.-H. Hsu, and Y.-W. Chang, "High-performance global routing with fast overflow reduction," *in 2009 Asia and South Pacific Design Automation Conference*, pp. 582–587, IEEE, 2009.

[4] Xu, Yue, Yanheng Zhang, and Chris Chu. "FastRoute 4.0: global router with efficient via minimization." *2009 Asia and South Pacific Design Automation Conference. IEEE*, 2009.

[5] Cho, Minsik, et al. "BoxRouter 2.0: Architecture and implementation of a hybrid and robust global router." *2007 IEEE/ACM International Conference on Computer-Aided Design. IEEE*, 2007.

[6] Ozdal, Muhammet Mustafa, and Martin DF Wong. "Archer: a history-driven global routing algorithm." *2007 IEEE/ACM International Conference on Computer-Aided Design. IEEE*, 2007.

[7] Gao, Jhih-Rong, Pei-Ci Wu, and Ting-Chi Wang. "A new global router for modern designs." *2008 Asia and South Pacific Design Automation Conference. IEEE*, 2008.

[8] Moffitt, Michael D. "MaizeRouter: Engineering an effective global router." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.11 (2008): 2017-2026.

[9] Roy, Jarrod A., and Igor L. Markov. "High-performance routing at the nanometer scale." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.6 (2008): 1066-1077.

[10] Zhang, Yanheng, Yue Xu, and Chris Chu. "FastRoute3. 0: a fast and high quality global router based on virtual capacity." *2008 IEEE/ACM International Conference on Computer-Aided Design. IEEE*, 2008.

[11] Pan, Min, and Chris Chu. "FastRoute: A step to integrate global routing into placement." *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design.* 2006.

[12] Pan, Min, and Chris Chu. "FastRoute 2.0: A high-quality and efficient global router." *2007 Asia and south pacific design automation conference. IEEE*, 2007.

[13] Liu, Jinwei, et al. "CUGR: Detailed-Routability-Driven 3D Global Routing with Probabilistic Resource Model."