Abstract— Processor schedules the processes, takes in job queue and places in ready queue. This placing is based on scheduling policies. To examine scheduling policies based on their mean response times. Another important, but sometimes opposing, performance metric is a scheduling policy’s fairness. For example, a policy that biases towards small job sizes so as to minimize mean response time may end up being unfair to large job sizes. We propose that all preemptive size based and age based policies are Always Unfair, but that remaining size based policies and non-preemptive policies are divided between two classifications. A scheme is being proposed to improve the schedulability of real-time embedded systems with fixed priority scheduling. Some systems prefer the higher priority job (HPJ) which have less CPU burst and better complexity.

I. INTRODUCTION

The performance of scheduling policies has been measured using mean response time and more recently mean slowdown [1-2]. Under these measures, size based policies that give priority to small job sizes at the expense of larger job sizes perform quite well. However, these policies tend not to be used in practice due to a fear of unfairness. Some examples have been examined just like web servers and scheduling in supercomputing centers based upon the same problem [3, 5]. To address the tension between minimizing mean response time and maintaining fairness, hybrid scheduling policies have also been proposed; for example, policies that primarily bias towards young jobs, but give sufficiently old jobs high priority as well [4, 6].

Unfairness problem have been seen by two researchers named “Bansal and Harchol-Balter” and they derived some sort of result which are in favor of the mean response time (MRT) for low and high priority jobs in average systems. DEFINITION 1.1. Jobs of the size $x$ are treated fairly under policy $P$ iff $E[S(x)]^P \leq 1/(1 - \rho)$. Further, a scheduling policy is fair iff it treats every job size fairly. DEFINITION 1.2. Jobs of size $x$ are treated unfairly under policy $P$ iff $E[S(x)]^P > 1/(1 - \rho)$. Further, a scheduling policy is unfair iff there exists a job size $x$ that is treated unfairly. A greedy approach proposed which executes all jobs by scheduling periodically according to their size and resources sharing priority which will reduce the resources and do their work fast. This extensive evaluation deduced by putting different workload on the system. Now resource shared are cached to reduce the scheduling more and more in the latest real time system [2-6].

II. RELATED WORK AND MOTIVATION

With these definitions, it is now possible to classify scheduling policies based on following

(i) Treat all job sizes fairly.
(ii) Treat some job sizes unfairly.
We define three types of unfairness [7];

A. ALWAYS FAIR
Processor sharing (PS) and Preemptive-Last-Come-First-Served (PLCFS) are the two main fair policies for scheduling. PLCFS always assign full processor to the most new process and have same result as PS. Equation derived for Always Fair method is following [2]

$$E[S(x)]^PS = E[S(x)]^{PLCFS} = 1 / (1 - \rho), \text{ for all } x$$

An important open problem not answered in this paper is the question of what other policies are in the Always Fair class. Question has received attention recently in the work of Friedman and Henderson, where the authors introduce a new preemptive policy, FSP that falls into this class. Although no queuing analysis of FSP is known, a simulation study suggests that it achieves performance similar to that of Shortest-Remaining-Processing-Time while guaranteeing fairness [8].

B. ALWAYS UNFAIR
Large number of common policies is Always Unfair. So, many common policies are guaranteed to treat some job size unfairly under all system loads. We will investigate classes of common policies, proving that each class is Always Unfair [2].

1) Non-size based, non-preemptive policies
When large job is in service cannot be preempted by a small job cause unfairness to small jobs. i.e Any non-preemptive policy $P$ is unfair for all loads under any service distribution defined on a neighborhood of zero [2].

2) Preemptive, size based policies
In this method a low priority job is always preempted by a higher priority job causing an unfairness to low priority jobs. i.e Preemptive-Shortest-Job-First (PSJF). A theorem proved for any preemptive, size based policy is Always Unfair [7-8].

C. FEEDBACK (FB)
In this method, the job with the least attained service gets the processor to itself. If several jobs all have the least attained service, they time-share the processor through PS. This is a practical policy, since a job’s age is always known, although its size may not be known [2-5].

D. AGE BASED POLICIES
Age based policies are interesting because they include many hybrid policies where, in order to minimize mean response time and curb the unfairness seen by large jobs, both sufficiently old jobs and very young jobs receive preferential treatment. Observe that under FB, priority is strictly decreasing with age. Thus, a new arrival will run alone until it achieves the age, $a$, of the youngest job in the system; and then those jobs of age will timeshare [9]. This timesharing is caused by the fact that if one job starts to run, its priority will drop, causing a different job to immediately run, and so on.
the case of a policy where priority is strictly increasing with age, a new arrival always has the lowest priority and can’t run until the system is idle [10].

A theorem age based policies are Always Unfair. The remainder of this section will prove this theorem using a method similar to the method used in Section 1.

We break the analysis into two cases:
(1) The case when there exists a finite sized job that has the lowest priority.
(2) When there is no finite sized job with the lowest priority.

E. SOMETIME UNFAIR
Sometimes unfair policies are those which behaves sometimes all job sizes fairly, but for other unfairly [2].

1) Non-preemptive, size based policies
It is based on the observation that if there is a lower bound on the smallest job size in the service distribution, then it is possible for a non-preemptive policy to avoid being Always Unfair. A theorem proved for this policy is “Any non-preemptive, size-based policy P is either Sometimes Unfair or Always Unfair” [2-3].

2) SRPT
In this at each moment of time, the server is processing the job with the shortest remaining processing time. It is well-known to be optimal for minimizing mean response time [11].

F. REMAINING SIZE BASED POLICIES
In this section examine the entire class of remaining size based policies (RSBP). The class of RSBP includes many hybrid policies; for example policies where, in order to minimize mean response time and curb the unfairness seen by large jobs, both jobs with very small and sufficiently large response times are given preferential treatment. In the same way as for age based policies, there are many possible mappings between priority and remaining size, allowing for multiple local minima in priorities and many interesting behaviors. A theorem proved for this policy is “All remaining size based policies are either Sometimes Unfair or Always Unfair” [2-9].

III. PROPOSED SCHEME AND SOLUTION
A technique is introduced to improve the schedulability of real-time embedded systems with fixed priority scheduling. Some systems prefer the higher priority job (HPJ) which have less CPU burst and HPJ preempts the lower priority job (LPJ), then LPJ append itself at the end of the ready queue, so some times starvation occurs, which is unfair with LPJ. Fixed priority scheduling means that the each job executes itself as their age in the ready queue, job may be either of lower priority or higher priority. Smallest number assigning to the job is the higher priority. Jobs like HPJ and LPJ share resources when they executes so CPU schedule the jobs little bit late because CPU allocates resources which will be shared among different jobs of different priorities. So, concept of greedy approach comes in mind which reduces the resource interference between higher-priority and lower-priority tasks, and thus enables more lower-priority tasks to be scheduled. Size based non-preemptive policies according to the age are used here. This is the closed form and ultimate solution to solve this kind to issue. Instead of this policy unfairness will occur while scheduling the HPJ and LPJ. Because both have different properties some HPJ has large size with high priority and some are of small size with less CPU burst and some LPJ has small size with low priority and some are of less CPU burst. A greedy approach [12] used which executes all jobs by scheduling periodically according to their size and resources sharing priority which will reduce the resources and do their work fast. According to research 20 to 35% deadline of tasks is reduced by system. This extensive evaluation deduced by putting different workload on the system shown in Table 1. Now resource shared are cached to reduce the scheduling more and more in the latest real time system.

A. IMPLEMENTATION AND COMPLEXITY EVALUATING
package practest;
import java.awt.Component;
import java.util.ArrayList;
import java.awt.Component;
package practest;
public class MainFrame extends javax.swing.JFrame {
    import javax.swing.JButton;
    public MainFrame extends javax.swing.JFrame {
        public MainFrame() {
            initComponents();
            // initializing a job with some resource
            for (Component component : comps) {
                if (component instanceof JButton) {
                    // for assigning priority to jobs according to resources
                    btnActionPerformed(java.awt.event.ActionEvent evt) {
                        temp = (JButton) evt.getSource();
                        String n = temp.getText();
                        String toks[] = n.split(" ");
                        int num = Integer.parseInt(toks[1]);
                        info = new ProcessInfo();
                        info.setDate(num);
                        info.setProcessNum(tot_btns);
                        readyQueue.add(info);
                    }
                    private ProcessInfo processInfo1;
                    private ProcessInfo processInfo2;
                    ArrayList<ProcessInfo> readyQueue = new ArrayList<>();
                    int tot_btns = 0;
                    Component comps[] = getContentPane().getComponents();
                    for (Component component : comps) {
                        if (component instanceof JButton) {
                            tot_btns++;
                            info = new ProcessInfo();
                            info.setDate(tot_btns);
                            readyQueue.add(info);
                        }
                    }
                    public MainFrame() {
                        initComponents();
                        readyQueue = new ArrayList<>();
                        int tot_btns = 0;
                        Component comps[] = getContentPane().getComponents();
                        for (Component component : comps) {
                            if (component instanceof JButton) {
                                tot_btns++;
                                info = new ProcessInfo();
                                info.setDate(tot_btns);
                                readyQueue.add(info);
                            }
                        }
                    }
                    public MainFrame() {
                        initComponents();
                        readyQueue = new ArrayList<>();
                        int tot_btns = 0;
                        Component comps[] = getContentPane().getComponents();
                        for (Component component : comps) {
                            if (component instanceof JButton) {
                                tot_btns++;
                                info = new ProcessInfo();
                                info.setDate(tot_btns);
                                readyQueue.add(info);
                            }
                        }
                    }
                }
            }
        }
    }
}

Ahmed Khan
Proposed Scheme and Solution
processInfo2 = readyQueue.get(j);  
if (processInfo1.getClicks() < processInfo2.getClicks()) {
    int num1 = processInfo1.getProcessNum();
    int num2 = processInfo2.getProcessNum();
    int t = num1;
    num1 = num2;
    num2 = t;
    processInfo1.setProcessNum(num2);
    processInfo2.setProcessNum(num1);
    readyQueue.set(i, processInfo2);
    readyQueue.set(j, processInfo1);
}

// for rendering results
String txt = "" +  
for (int i = 0; i < readyQueue.size(); i++) {  
    txt += processInfo.getClicks() + "      " + processInfo.getProcessNum() + "      " + processInfo2.getClicks() + "      ";
    JLabel3.setText(" Executing Process " + readyQueue.get(i).getProcessNum() + " consuming " + readyQueue.get(i).getClicks() + " resources.");
}

Table 1. Complexity sorting performed for each job

<table>
<thead>
<tr>
<th>Job</th>
<th>Iterations of sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>n</td>
<td>n-1</td>
</tr>
<tr>
<td>Jobs = ( \sum_{i=1}^{n} i )</td>
<td>Steps = ( \sum_{i=1}^{n-1} i )</td>
</tr>
</tbody>
</table>

1) Best Case and Average Case

If job is single or the jobs have equal or fair resources and already sorted then the best case would be \( \Theta(1) \).

2) Worst Case

If jobs have unfair resources and completely unbalanced in the ready queue then each iteration losses the sorting step by \( (n-1) \) for each job, so the worst case would be \( \Theta(n^2) \).

IV. CONCLUSION

Aim of this paper is to understand unfairness. From this research it’s obvious that unfairness is a function of load and fair to all jobs. Yet at higher loads, these policies become unfair. Three classifications of scheduling policies: Always Unfair, Sometimes Unfair, and Always Fair. Rather than classifying individual policies, we group policies into different types: size based, age based, remaining size based, and others. The result that all preemptive size based policies are Always Unfair may seem surprising in light of the fact that one could choose to assign high priority to both small jobs and sufficiently large jobs in an attempt to curb unfairness.

V. REFERENCES