CHAPTER 4

WINDOW-CONSTRAINED SCHEDULING
FOR CPU TASKS

4.1 INTRODUCTION

In the relaxed version of the window-constrained problem, job instances may be serviced after their deadlines, as long as a job receives minimum fraction of service in fixed time intervals. For this problem, Zhang et al (2004 a) proposed VDS. GA based approaches are able to optimize multiple objectives simultaneously.

4.2 PROBLEM FORMULATION

Consider Jn independent user jobs n= {1, 2, 3 ... n} on the resource in a weakly hard real-time system. It is assumed that the jobs are periodic. For scheduling the jobs, each job has service constraints in terms of a request period T, a window-constraint m/k and a service quantum C. The window-constrained job, Ji, is defined by a 4-tuple (C_i, Ti, m_i, k_i).

The request period (T_i) of a job instance Ji is the interval when it is ready and when it must complete service for a specific amount of time. The end of one request period and the start of the next, denote a deadline by which time a job J_i must be serviced for one quantum of C_i time units. A schedule is said to be feasible if m_i instances are serviced out of k_i instances of a job in every non overlapping window of k * T time units.

One of the problems in CPU scheduling is context switch; one process needs to be switched out of the CPU so another process can run.
Switching from one process to another requires a certain amount of time and thereby reduces the performance of CPU.

A window-constrained schedule, T: T1, T2 … Tn is called feasible if all real-time jobs, Ji for all (i: 1 to n) satisfy that a minimum of mi out of ki consecutive job instances must be serviced for Ci time in every window time of ki*Ti for each job Ji with request period Ti. The minimum utilization factor required for a set of n periodic jobs is calculated as given in Equation (4.1).

\[ U_{\text{min}} = \frac{\sum_{i=1}^{n} m_i * c_i}{k_i * T_i} \]  

(4.1)

One of our objectives is to maximize \( U_{\text{min}} \) by maximizing the number of jobs meeting the deadline. The other objectives are to minimize the number of service delays and the number of context switches simultaneously.

4.3 PROPOSED SOLUTION STRATEGIES

In this section, a method for window-constrained scheduling based on GA approaches is proposed. In the approaches elitism is included to retain the best individuals. At first population initialization is introduced. A new coding scheme is proposed and evaluated. Afterward, a penalty function which is utilized in the fitness function is introduced. A new crossover and mutation operators are described.

4.3.1 MOGA

The pseudo code for MOGA is discussed in Section 3.3.1, given by Figure 3.7.
Population Initialization

At the beginning, the algorithm randomly generates the chromosomes. This algorithm gets the number of jobs as input. For each job, C, T, m and k values are generated randomly such that C <= T and m <= k. The window size for each job is calculated using Equation (4.2).

\[
\text{win\_size} = k \times T
\]  

where \( \text{win\_size} = \text{window size} \)

The population is initialized randomly. The population (pop) is an array that its \( i^{th} \) row shows the \( i^{th} \) chromosome, and the \( j^{th} \) column of the \( i^{th} \) row shows the \( j^{th} \) genome in \( i^{th} \) chromosome. The \( \text{rand} \) (N) is a function that returns a random integer in the interval \([0, N-1]\). The structure of chromosomes is presented in the next sub-section. Figure 4.1 shows how the jobs are generated.

<table>
<thead>
<tr>
<th>Input: Number of jobs (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Pop (Initialized population)</td>
</tr>
</tbody>
</table>

Consider a two dimensional array Job that has four columns and \( N \) rows

\[ \text{for } i: 0 \text{ to } N-1 \]
\[ \text{for } j: 0 \text{ to } 3 \]
\[ \text{Job}[i][j] = \text{rand}(N); \]

Check if C <= T and m <= k

Consider a two dimensional array Pop that has win_size columns and popsize rows

\[ \text{for } i: 0 \text{ to } \text{popsize} - 1 \]
\[ \text{for } j: 0 \text{ to } \text{win\_size} - 1 \]
\[ \text{Pop}[i][j] = \text{rand}(N); \]

Return Pop

Figure 4.1 Algorithm for generating the job sets
Coding scheme

In this work, the candidate solutions in the population have a direct representation. A chromosome represents the job sequence in which the various jobs are to be processed. The chromosome is an array of n integers, where n is the total number of jobs to be scheduled. The allele value at the ith entry of a chromosome represents one instance of a job. The gene represents the job id which is an integer. Table 4.1 shows an example set which consists of 2 jobs. C, T, m, k values for the jobs are tabulated. For this example, the representation of an individual is shown in Figure 4.2. The maximum window size is calculated by Max of \((k_i \times T_i)\) for each job \(J_i\). The length of an individual is equal to the \(k_i \times T_i\). (i.e.) \(3 \times 3 = 9\). Thus the number in ith gene position denotes the time slot at which the job instance of a job, \(J_{ij}\) is serviced. (i.e.) at time slot 5, 5th instance of job 1 is serviced.

Table 4.1 Example Set 1

<table>
<thead>
<tr>
<th>Job</th>
<th>C</th>
<th>T</th>
<th>m</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>(J_1)</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(J_2)</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccc}
J1 & J1 & J1 & J1 & J2 & J2 & J2 & J2 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

↑ Job instances with job ids  ↑Gene

Figure 4.2 The gene representation for a job instance

Fitness function

In this sub-section the proposed fitness function is presented. In window-constraint scheduling problem, the goal is to find an order of executing jobs such that the number of job instances serviced within the window is to be maximized. Simultaneously, the conflicting objectives, the
number of service delays and the number of context switches are to be minimized. In this way, a penalty function has been designed which guarantees the best minimum utilization factor. In this function, with increasing number of delays and number of context switches, the penalty value is increased such that the value of fitness becomes different for the chromosomes with the same number of missed deadlines. Equation (4.3) shows the formula that can be used to evaluate the fitness of each chromosome.

\[
f(W, S, D) = \alpha \sum_{i=1}^{n} W_i - (\beta S + \gamma D)
\]

where
- \( W \) - weight for meeting the deadline.
- \( W_i \) - \{0, if deadline is not met; 1, otherwise\}
- \( S \) - Number of context switch for a schedule.
- \( D \) - Number of Service delay experienced by a schedule.
- \( n \) - Number of jobs.
- \( \alpha, \beta, \gamma \) - coefficients for weight, context switch and delay.
- \( \alpha \in \mathbb{Z}^+, \alpha = 100 \).
- \( \beta = 10\% \) of \( \alpha \).
- \( \gamma = 1\% \) of \( \alpha \).

As Equation (4.3) shows, the fitness function is maximized when the number of context switches and delays are zero and more number of job instances has been serviced. Weighted formula approach is used in designing fitness function. Each objective is assigned a weight and converted into a single equation. The weights are either 0 or 1. The objective of maximizing the \( U_{\text{min}} \) factor is obtained through maximizing the number of jobs serviced within the window. This objective is assigned the highest weightage, which is achieved through assigning randomly 100 to \( \alpha \); simultaneously, the number of context switches and delays are to be minimized. Number of context switches is another important parameter in CPU scheduling that affects the
performance. Hence coefficient of number of context switch, $\beta$ is assigned the value, 10% of $\alpha$. $\gamma$ is assigned a value, 1% of $\alpha$.

**Genetic Operators**

In the implementation, the single point crossover with probability $P_c$ has been used to combine chromosomes and prepare offspring. Furthermore, the uniform mutation method with probability $P_m$ has been used as mutation operator and elitism has been considered as a selection procedure.

**Stopping Criteria**

Fitness calculation, sorting, selection, crossover and mutation procedures are applied for 30 generations to get convergence. In most of the runs, convergence occurs between 26th and 30th generation. Hence, reaching 30th generation is the stopping criteria.

### 4.3.2 HGA

Though GA is capable of finding optimal solutions, it traps into local optima sometimes. To overcome this drawback, GA is combined with local search techniques such as SA and Tabu search. As explained in Section 3.3.2, in this work GA is hybridized with SA.

### 4.3.3 Micro-GA

Micro-GA is a genetic algorithm which works on a very small population and a reinitialization process. It is described in Section 3.3.4 and the structure of micro-GA is illustrated in Figure 3.14. During the execution of micro-GA, the nondominated vectors are stored in an external memory. When nominal convergence (low number of generations 2 to 5 or similarities among the strings) is achieved, two nondominated vectors from the final
population are selected and compared with the contents from the external memory. If either of them (or both) remains as nondominated then they are included in the external memory. These two are also compared with two elements (randomly selected) from the replaceable portion of the population memory. If either of these vectors dominates, then population memory vector is replaced by it. As micro-GA goes through many cycles, the replaceable memory will tend to have more nondominated vectors. Some of them will be used in the initial population of the micro-GA to start new evolutionary cycles.

**Micro-Pop Initialization**

In this work, the micro population size is taken as 6. Three solutions are selected from the population memory randomly and saved as the first three solutions in micro population. To keep diversity, the other three solutions are selected randomly from other part of the population memory.

Figure 4.3 gets the number micro-population size as input. It generates the micro population, a two dimensional array. The window size (win_size) of the schedule in the array is the chromosome length of the solutions. First half of the micro population (micro-pop) contains the solutions selected randomly from the initial population. It is known as replaceable memory. Its size is 50% of the micro-population size. The second half of the micro population, non replaceable memory, is filled with solutions randomly selected from the initial population.
**Input:** micro-population size (micropop_size)

**Output:** micro-pop (Initialized micro-population)

Consider a two dimensional array micro-pop that has win_size columns and micropop_size rows

for i: 1 to micropop_size/2
    k = rand (popsize/2)
for j: 0 to win_size - 1
    micro-pop[i][j] = Job[k][j]
for i: micropop_size/2+1 to micropop_size
    r = rand(popsizesize)
    if r < micropop_size/2, then r = r + micropop_size/2
for j: 0 to win_size - 1
    micro-pop[i][j] = Job[r][j]
return micro-pop

**Figure 4.3 Algorithm for generating the micro-population**

The fitness function designed in Section 4.3.1 is used in this approach. It guarantees the best minimum utilization factor.

This algorithm depicts the method by which the number of job instances that are serviced in a schedule (W) is found out. Micro population is the input. For each job, the number of job instances to be serviced in a window is calculated using service time (C) and number of minimum deadlines of the job (m). In a window, if all the instances of a job are serviced, then W is incremented. For each and every job, W is found out using the algorithm shown in Figure 4.4.
**Input:** micro-pop

**Output:** Number of job instances serviced for all jobs for each schedule in the micro-pop, W

Consider a two dimensional array micro-pop that has win-size columns and micropop_size rows

for i: 0 to micropop_size - 1

for k: 0 to number of jobs - 1

    win = C[k] * m[k]; count=0;

    for j: 0 to win_size-1

        if (micro-pop[k][j] != job[k]) count++;

    if (win != count) && (win>count) then

        W[k]=count; else W[k]=0;

    return W[k];

**Figure 4.4 Algorithm for number of job instances serviced**

Figure 4.5 illustrates the algorithm to find out the number of context switches, S made for each schedule. It gets the micro population as the input. In each schedule, if there is a change of job instance, then a context switch occurs. For all solutions in the micro population, the number of context switches is found out.

**Input:** micro-pop

**Output:** number of context switches for all the schedules in the micro-pop, S

Consider a two dimensional array micro-pop that has win_size columns and micropop_size rows

for i: 0 to micropop_size - 1

for j: 0 to win_size – 1

    if (micro-pop[i][j] != micro-pop[i][j+1])

        S[i]++;

return S;

**Figure 4.5 Algorithm for finding the number of context switches**
Figure 4.6 shows how the maximum service delay for a schedule is found out (D). For this, algorithm takes the micro population as input. In each schedule, it finds the service delay experienced by each job. The delays of all the jobs (i.e. curr_delay) are compared. The maximum value is the maximum service delay experienced by the schedule. It is found for all individuals in the micro population.

\begin{minipage}{\textwidth}
\begin{verbatim}
Input: micro-pop
Output: Maximum service delay experienced by the jobs in the schedule in the micro-pop (D)
Consider a two dimensional array micro-pop that has win_size columns and micropop_size rows
for (i: 0 to micropop_size - 1)
{ D[i] = 0;
  for j: 0 to N - 1
  { Job = j; max = 0;
    for k: 0 to win_size-1
    { if (micro-pop[j][k] !=job)
      then { curr_delay++;
        if (curr_delay > max) then max [j] = curr_delay;
      } else curr_delay = 0; }
    if (max[j] > D[i]) then D[i] = max[j];
  }
  return D;
}
\end{verbatim}
\end{minipage}

\textbf{Figure 4.6 Algorithm for finding the maximum service delay}

Figure 4.7 shows how first form of elitism is applied. esize is an important parameter which denotes the solutions retained in the micro population. The best individuals (buffer) are passed to the next micro-
generation. The number of individuals is decided by the factor esize. In this work, esize value is 2.

| Input: esize, The number of best individuals to be retained by micro population |
| Output: Buffer, Individuals retained by micro population |
| for i: 0 to esize - 1 |
| for j: 0 to win_size - 1 |
| buffer[i][j] = micro-pop[i][j] |
| return buffer |

**Figure 4.7 Algorithm for first form of elitism**

At a fixed number of iterations some of the non-dominated solutions from the external memory are used to update the replaceable portion. The iteration is selected by the parameter elite3_rate which is set as 30 in this work. The individuals are selected randomly by using rand function and are replaced by the individuals selected from the external memory. The method is third form of elitism shown in Figure 4.8.

| Input: Number of generation |
| Output: Nondominated solutions from the external memory |
| k = rand (popsize/2) |
| for i: 0 to Number of generation - 1 |
| for j: 0 to win_size - 1 |
| Job[k][j] = ext-mem[i][j] |
| return Job |

**Figure 4.8 Algorithm for third form of elitism**

The representative solution of the micro-GA cycle replaces a randomly selected individual of the replaceable portion if it wins the tournament.
Figure 4.9 depicts how second from of elitism is applied. Randomly selected individuals in the initial population are replaced with the individuals selected (Job) from the micro population. This algorithm gets the number of individuals as input and returns the replaced individuals in the initial population as output.

### Input: Selected individuals in the initial population

### Output: Selected solutions from the micro population

```
k = rand (popsize/2)
for i: 0 to Number of generation - 1
    for j: 0 to win_size - 1
        Job[k][j]=micro-pop[i][j]
return Job
```

**Figure 4.9 Algorithm for second form of elitism**

**Stopping criteria**

When there is not much improvement in the fitness value from generation to generation, the algorithm stops.

### 4.4 IMPLEMENTATION AND RESULTS

In this sub-section, experimental results are reported and they are compared with related works. To do comparison precisely, several methods with different size of job instances were implemented.

**Parameter setting**

The initial population was taken as 200. Mutation rate and Crossover rate were set as 10% and 90% respectively. The parameter esize decides the number of solutions in the micro population passed to the next generation without alteration. The work was done with 1, 2 and 4 as esize. The experiment was run 500 times for each esize. It was found that the
percentage of jobs serviced without missing the deadline are 78, 99 and 80 for the esize 1, 2 and 4 respectively. It is shown in Figure 4.10 that when the esize was set as 2, the window-constraint violation was only 1%. Since, esize 2 gives a better performance of the proposed approach the esize was fixed at 2 for all further tests.

Table 4.2 A randomly generated job set

<table>
<thead>
<tr>
<th>Job</th>
<th>C</th>
<th>T</th>
<th>m</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₁</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>J₂</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>J₃</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

This job set is taken as an example for fixing the micro-population size. The job set is given Table 4.2. Krishnakumar (1989 a) and Coello and Pulido (2001) set the micro-population size as 5 and 4 respectively. In this work, micro-population size is set at various values (i.e. 3, 4, 5, 6, and 7) and the corresponding fitness value, converging generation and the number of missed tasks are found out. The results are shown in Table 4.3. It shows that at size 3, the highest fitness value (i.e. 45) is reached. Convergence also occurs at 7th generation. However, number of missed tasks is high (i.e. 8). Setting micro-population size as 6 provides a fitness value 35. But the performance is better regarding convergence and number of missed tasks. Hence, throughout the experiment, the micro-population size is fixed as 6.

Table 4.3 Comparison of micro population sizes

<table>
<thead>
<tr>
<th>Micro population Size</th>
<th>Fitness value reached</th>
<th>Converging generation</th>
<th>Number of missed tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>45</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>35</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>35</td>
<td>14</td>
<td>6</td>
</tr>
</tbody>
</table>
The parameter elite rate decides the probability of applying third form of elitism. The test is conducted for the elite rate values 5, 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 percentage. For each elite rate, the highest fitness values reached and the converging generations were found. The results are shown in Figures 4.11 and 4.12.
At elite rate 30%, the fitness value reached, 1252 is the highest value. However, the convergence occurs only at 23\textsuperscript{rd} generation for 30% elite rate whereas other elite rates showed better performance. For example, at 40%, the convergence occurs at 15\textsuperscript{th} generation. But, there is no much appreciable difference in the time taken for execution. Hence, for this work, elite rate was selected as 30%.

**Simulation Environment**

$C_i$, $T_i$, $m_i$, $k_i$ for 500 job sets were generated randomly according to the constraint $U_{\text{min}} \leq 1$ for each job (i.e.) 2 jobs, 3 jobs, 4 jobs, 5 jobs, 10 jobs and 15 jobs.

The experiment was performed 100 times with different random generator seed at each run. In these runs, 200,000 jobs had been given to the algorithms. It was assumed that all jobs in each job sequence are periodic with unit processing time. The jobs may have different request periods. Each job $J_i$ had a new instance with every request period, $T_i$. A scheduling decision was made once every unit-length time slot, $\Delta$. Scheduling was performed for each
job sequence to capture all possible window-constraint violations. The initial population size was fixed as 200. Elitism rate was selected as 30%. The micro population size is 6. Three forms of elitism were applied. The test had been conducted for 2 jobs, 3 jobs, 4 jobs, 5 jobs, 10 jobs and 15 jobs. The proposed function in Equation (4.3) has been considered as a fitness function and to make the results comparable with other methods, the priority of all tasks are considered same.

All implementations were performed on a personal computer with 2.26 GHZ of CPU and 3 GB of RAM in the C environment.

The experimental results showed a significant improvement in window-constrained scheduling using GA approaches in comparison with VDS.

Table 4.2 shows the parameters used in this work. The parameters have been already discussed in the previous sections.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>200</td>
</tr>
<tr>
<td>Elitism rate</td>
<td>30%</td>
</tr>
<tr>
<td>Crossover</td>
<td>Single Point Crossover</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>90%</td>
</tr>
<tr>
<td>Mutation</td>
<td>Uniform Mutation</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>10%</td>
</tr>
<tr>
<td>Micro Population Size</td>
<td>6</td>
</tr>
<tr>
<td>Forms of Elitism</td>
<td>3</td>
</tr>
</tbody>
</table>

Optimal Schedules with optimal window-constraint, minimal service delay and minimal context switches.
The schedules were produced by the proposed approach for 2, 3, 4, 5, 10 and 15 jobs. The jobs are generated for both $U_{\text{min}} < 1$ and $U_{\text{min}} = 1$. It was run for 1000 times for each job. Thus, 78,000 jobs were generated totally. Results are given in Figure 4.13, 4.14 and 4.15.

![Graph showing comparison of % of missed deadlines](image)

**Figure 4.13 Comparison of % of missed deadlines**

From Figure 4.13, it is found that for the GA based approaches, percentage of missed deadlines is less than that of VDS. When the numbers of jobs are 2 or 3, the missed deadlines in the schedules are less. But, if the number of jobs is increased, more job instances are missed. For example, when 15 jobs were scheduled, only 8% of the job instances are serviced without missing the deadline. On an average, 15% of the job instances are serviced without missing the deadline when GA based approaches are used. Schedules produced by micro-GA have less percentage of missed deadlines than that of MOGA and HGA.
The average of the service delays for each job is illustrated in Figure 4.14. It is shown that as the number of jobs increases, delays for the jobs also increase for both VDS and GA based approaches. However, in the results produced by the GA based approaches, there is a less number of service delays. 17%, 18.7% and 18.7% improvement in the performance of the GA are shown by MOGA, HGA and micro-GA respectively. Percentage of average delay is marginally low for micro-GA when compared with other GA approaches.

![Figure 4.14 Comparison of % of average service delay](image)

Figure 4.14 Comparison of % of average service delay

When the percentage of average context switches is compared, the proposed approach outperforms VDS. The results are shown in Figure 4.15. On an average, there were 3.5, 4.8, 4.7 and 5.8 context switches in the schedules produced by MOGA, HGA, micro-GA and VDS respectively.
Figure 4.15 Comparison of % average context switch

Schedules produced for jobs with unequal service time by VDS and proposed approach when $U_{\text{min}} < 1$

Figure 4.16 shows an example for which VDS violates the window-constraint even though the utilization factor $U_{\text{min}} < 1$. In this example, 4 jobs are considered with C, T, m and k values. The service times of the jobs are not equal. The request periods for jobs 2 and 3 are not the multiples of C2 and C3 respectively. The minimum utilization factor $U_{\text{min}}$ is 0.9375 [2* 2 / 4 * 4 + 3 * 1 / 4 * 4 + 3 * 2 / 4 * 4 + 1 * 2 / 4 * 4].

<table>
<thead>
<tr>
<th>Job</th>
<th>C</th>
<th>T</th>
<th>m</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>J2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>J3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>J4</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 4.16 Example job set for which VDS violates the window-constraint though $U_{\text{min}} < 1$
A minimum of 4 job instances are to be serviced for J1 within the window 16. 3, 6 and 2 job instances are to be serviced for J2, J3 and J4 respectively.

![Figure 4.17 Schedule produced by VDS when $U_{\text{min}} < 1$](image)

The schedule produced by VDS is illustrated in Figure 4.17. VDS is able to service 4 instances of J1, 5 instances of J2 and 6 instances of J3. However, it is not able to service all the two instances of J4 but services only 1 instance of J4. It is understood that VDS violates the window-constraint. It is not able to produce a feasible schedule though $U_{\text{min}}$ is less than 1.

Figures 4.18, 4.19 and 4.20 show the schedule produced by the proposed approaches. The proposed approaches are capable of servicing 4, 3, 6 and 2 job instances of J1, J2, J3 and J4 respectively. The minimum requirement is satisfied by the schedule. Even though the service times of the jobs are different and the request periods $T_2$, $T_3$ are not the multiples of service times $C_2$ and $C_3$ respectively, the proposed approaches produce a feasible schedule.

![Figure 4.18 Schedule produced by MOGA when $U_{\text{min}} < 1$](image)

![Figure 4.19 Schedule produced by HGA when $U_{\text{min}} < 1$](image)

![Figure 4.20 Schedule produced by micro-GA when $U_{\text{min}} < 1$](image)
Schedules produced for jobs with unequal service time by VDS and proposed approaches

Figure 4.21 illustrates an example for which VDS violates the window-constraint even though the utilization factor $U_{\text{min}} = 1$. In this example, three jobs J1, J2 and J3 are considered. According to the values C, T, m and k, Job 1(J1) has 3 instances with a service time 2 to be serviced. 3 instances of J2 are to be serviced with service time 1 and 1 instance of J3 is to be serviced with service time 3. The service times of J1, J2 and J3 are 2, 3 and 1 respectively (i.e.) C1, C2 and C3 are not equal. The request period of J1 (T1) is not a multiple of service quantum of J1 (C1). The minimum utilization factor, $U_{\text{min}}$ for this job set is found to be 1 (i.e.) $2 \times 3 / 3 \times 4 + 3 \times 1 / 3 \times 4 + 1 \times 3 / 3 \times 4$. The schedules produced by VDS and proposed approach are given in Figs 13 and 14 respectively.

<table>
<thead>
<tr>
<th>Job</th>
<th>C</th>
<th>T</th>
<th>m</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>J2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>J3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

According to the window-constraint for J1, out of 12 job instances, a minimum of 6 job instances are to be serviced. For J2, at least 3 out of 12 job instances are to be serviced. For J3, at least 3 out of 12 job instances are to be serviced. VDS services 6 and 3 job instances for J1 and J2 respectively. However, VDS can service only 1 out of 12 job instance of J3. So, 2 instances
of J3 are not serviced by VDS. As VDS cannot handle the jobs having varying service times and request period, only 1 job instance is serviced which violates the window-constraint of J3. In this example, though $U_{\text{min}}$ is equal to 1, J1 and J3 have different service times. The schedule produced by VDS is illustrated in Figure 4.22.

![Figure 4.22 Schedule produced by VDS when $U_{\text{min}} = 1$](image1)

![Figure 4.23 Schedule produced by MOGA when $U_{\text{min}} = 1$](image2)

![Figure 4.24 Schedule produced by HGA when $U_{\text{min}} = 1$](image3)

![Figure 4.25 Schedule produced by micro-GA when $U_{\text{min}} = 1$](image4)

Figures 4.23, 4.24 and 4.25 show the schedules produced by MOGA, HGA and micro-GA respectively. The proposed approaches service all the 6, 3 and 3 job instances of J1, J2 and J3 respectively. In this result, it is shown that the window-constraints of all the jobs are satisfied. Even though the service times of the jobs are different ($C_1! = C_2$) and the request period ($T_1$) is not the multiple of service time ($C_1$), proposed approach produces a feasible schedule.

![Figure 4.25 Schedule produced by micro-GA when $U_{\text{min}} = 1$](image5)

**Rate of feasible Schedules utilizing 100% of the resources**

$C$, $T$, $m$, $k$ values for jobs 2, 3, 4, 5, 10 and 15 are generated randomly such that minimum utilization factor is equal to 1 ($U_{\text{min}} = 1$). The result is illustrated in Figure 4.26 which shows that the percentage of feasible schedules utilizing 100% resources produced by the proposed approaches is higher than that of VDS. The proposed approaches perform better than VDS.
However, as the number of jobs increases then the rate of feasible schedules is very much reduced.

![Figure 4.26 % of feasible schedules produced](image)

**Pareto Front produced by proposed approaches**

In this work, the objectives minimizing the number of context switches and number of service delay are contradictory. A trade-off between the objectives is found out by conducting experiments for the proposed approaches by keeping the values of service delay as 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14. Corresponding number of context switches are noted. Pareto front produced by MOGA, HGA and Micro-GA are shown in Figure 4.27.
Number of calculations involved in proposed GA approaches

The performance and the efficiency of the various GAs can be compared under two criteria.

- Fitness value reached and Number of calculations involved

An example set with four processes is considered. C, T, m, k values are given in Table 4.3.

<table>
<thead>
<tr>
<th>Process id</th>
<th>C</th>
<th>T</th>
<th>m</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

For this example MOGA, HGA and micro-GA are applied. The schedule produced by MOGA is shown in Figure 4.27.
The fitness value that we have attained is 1142. The number of GA function (selection, cross over and mutation) evaluations is 2400. This value is attained by the following calculations.

Number of function evaluations = population size * Converged generation

\[ = 300 \times 8 = 2400 \]

<table>
<thead>
<tr>
<th>J4</th>
<th>J2</th>
<th>J1</th>
<th>J1</th>
<th>J1</th>
<th>J1</th>
<th>J3</th>
<th>J3</th>
<th>J2</th>
<th>J2</th>
<th>J2</th>
<th>J2</th>
<th>J2</th>
<th>J2</th>
<th>J3</th>
<th>J3</th>
<th>J3</th>
</tr>
</thead>
</table>

**Figure 4.29 Result of HGA**

The schedule produced by HGA is illustrated in Figure 4.28. The fitness value that we have attained is 1143. The number of GA function (selection, cross over and mutation) evaluations is 1700. This value is attained by the following calculations.

Number of function evaluations = population size * Converged generation

\[ = 50 \times 34 = 1700 \]

| J4 | J3 | J3 | J2 | J2 | J2 | J2 | J2 | J1 | J1 | J3 | J3 | J3 | J1 | J1 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

**Figure 4.30 Result of micro-GA**

Figure 4.29 shows the schedule produced by micro-GA. It converges at the fitness value 1141. The number of GA function (selection, cross over and mutation) evaluations is 1800 which is lesser than MOGA calculations. This value is attained by the following calculations.

Number of function evaluations = Micro pop size * Inner cycles * Converged gen

\[ = 6 \times 12 \times 25 = 1800 \]

The comparative results are tabulated in Table 4.4.

<table>
<thead>
<tr>
<th>GA used</th>
<th>Number of function evaluations</th>
<th>Fitness value at convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOGA</td>
<td>2400</td>
<td>1142</td>
</tr>
<tr>
<td>Micro-GA</td>
<td>1800</td>
<td>1141</td>
</tr>
<tr>
<td>HGA</td>
<td>1700</td>
<td>1143</td>
</tr>
</tbody>
</table>
From the results, it is found that fitness value at convergence is almost equal for the GA based approaches. However, the number of calculations is very less for HGA. Micro-GA’s performance is marginally equal to that of HGA. MOGA fails to produce a good performance.

4.5 SUMMARY

Window-constrained scheduling for CPU tasks is formulated as multi-objective optimization problem. MOGA, HGA and micro-GA are implemented. From the experimental results, it is found that GA based approaches outperform VDS in optimizing the schedule. Micro-GA shows marginal improvement in performance over MOGA and HGA in optimizing the schedules. However, the number of calculations involved in implementing of micro-GA is the limitation.