Optimal Selection of Preemption Points to Reduce Preemption Overhead

Presented by:

Kethu Sesha Sarath Reddy (2020102028)

Venkannagari Shirisha Reddy (2020102054)

Problem Statement

- Considering real time systems, we need to have proper schedulability analysis which needs accurate evaluation of execution times.
- Execution time of task is dependent on the task code and input. Especially for the preemptive tasks, execution time highly effected by preemption overheads. So, it is important to have proper estimation of execution time of preemptive tasks.
- For this purpose, there are timing tools but, most of them have pessimistic considerations leading to lose bound for WCET.
- This paper mainly focuses on reducing preemption overheads by selecting optimal locations at which preemption can occur which in turn leads to decrease in WCET. Thus, increasing schedulability of the task set.

System Model

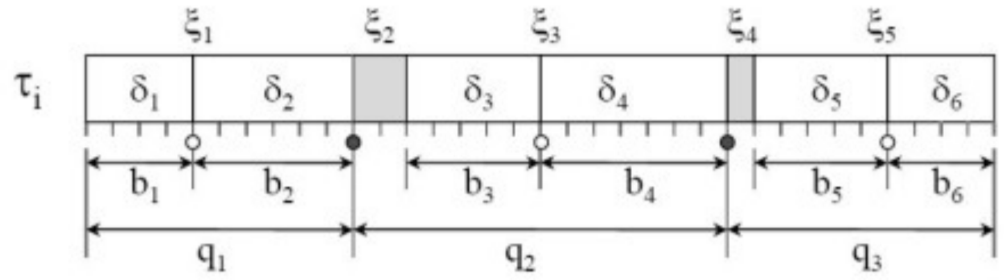
Terminology used:

- ➢ Consider **set t** of n periodic and sporadic real time tasks. Each **task t_i** generates infinite sequence of jobs with minimum inter-arrival time <= T_i
- ➢ Each t_i is divided into **N_i** non preemptive **basic blocks (BBs)** with **N_i-1 Potential Preemption Points (PPPs)** between every consecutive BBs.
- ➢ PPPs are the points where preemption may or may not occur.
- ➢ **Effective Preemption Points (EPPs)** are the points where preemption occurs to reduce **Cache Related Preemption Delay (CRPD)** and context switch time.
- ➢ Regions between each EPPs pair forms **Non-Preemption Region(NPR)**.

System Model

- Notations used:

- C_i^{NP}: Estimated WCET when the task t_i is executed in a fully non preemptive mode.
- C_i: Estimated WCET when the task t_i is executed preemptively.
- T_i: Period (for periodic tasks) or minimum inter-arruval time (for sporadic task).
- D_i: Relative deadline of t_i.
- $\delta_{i,k}$: k^{th} BB of t_i.
- $b_{i,k}$: Length of $\delta_{i,k}$ when task t_i is executed non-preemptively.
- p_i: Number of NPRs of task t_i determined by p_i-1 EPPs selected by algo.
- $q_{i,j}$: WCET of jth NPR of t_i including preemption cost.
- $q_i^{max} = \max\{q_{i,j}\}_{j=1}^{pi}$
- $\varepsilon_{i,k}$: worst case preemption overhead introduced when preemption takes place at kth PPP.



Theory Proposed

- Schedulability can be increased by using suitable scheduling algorithms to limit number of preemptions as much as possible and to reduce preemption overhead.
- General approaches like fully preemptive model create problems regarding overhead and WCET analysis and fully non-preemptive model imposes large no of blockings on higher priority tasks which may result in deadline misses.
- Thus, hybrid preemption strategies are adopted one such model is applied to reduce preemption overhead.
- Hybrid preemption strategy used in this problem includes deferring the preemption request by higher priority task until a point which results in less CRPD is reached (EPPs).
- Core idea of selecting these EPPs is discussed.

Proposed Approach

$$q = \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell,$$

$$B_k = B_{j-1} + q = B_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell.$$

- Let the maximum time for which task T_i can execute non preemptively be Q_i units.
- WCET of any NPR consisting $\delta_j, \delta_{j+1}, \ldots \delta_k$ is q. For feasibility $q < Q$
- Let B_k be WCET of first k BBs, including preemption overhead if there is EPPs in first k BBs.
- Therefore, with selection of EPPs, the possible WCET of the task T_i is B_N.

Proposed Approach

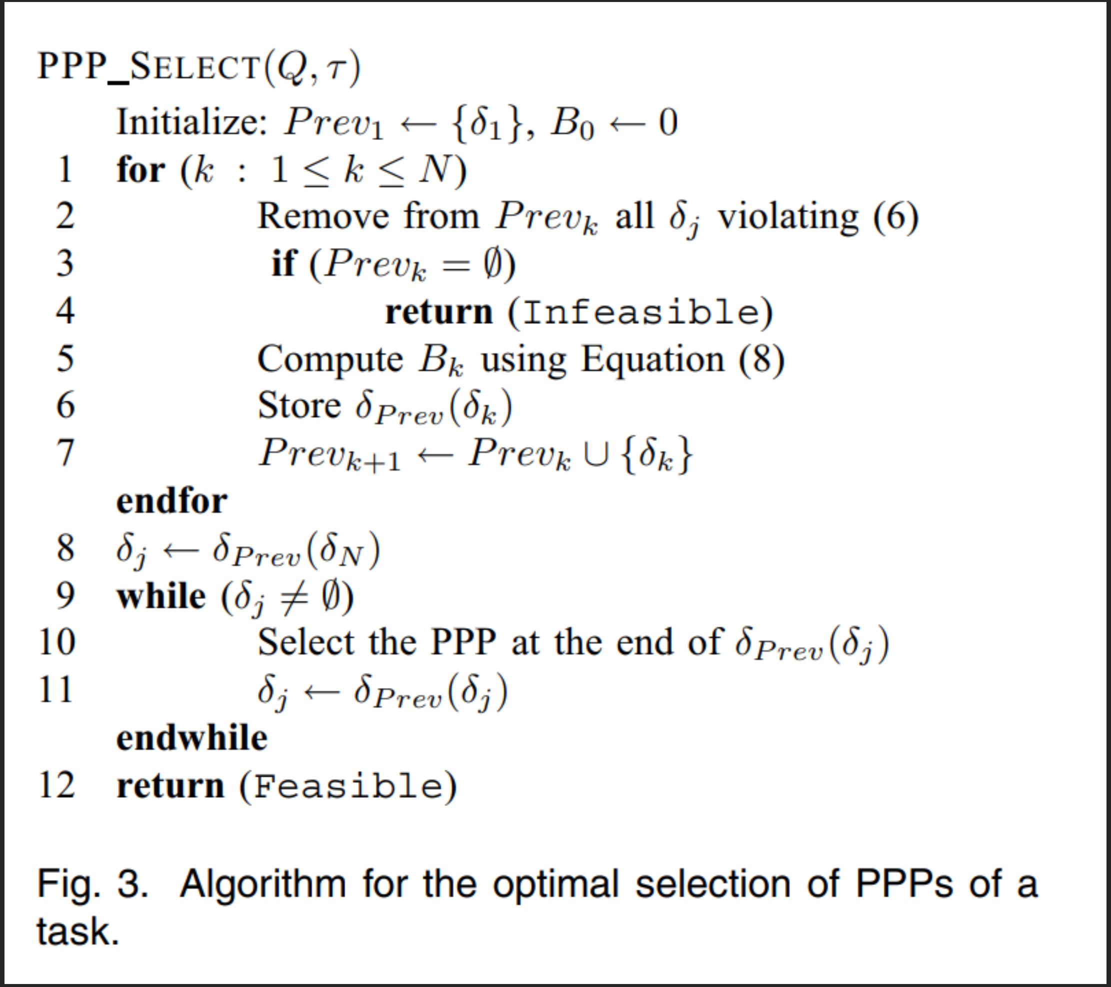
- Let $Prev_k$ be defined as the set of preceding BBs $\delta_j <= k$ that satisfy equation. $\xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \leq Q.$ - eq1

- Optimal selection of EPPs leads to minimum possible of B_ks thus we have $B_k = \min_{\delta_j \in Prev_k} \left\{ B_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}$ -eq2 $\delta^*(\delta_k) = \arg\min_{\delta_j \in Prev_k} \left\{ B_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}$

- Let $\delta_{Prev}(\delta_k)$ be the basic block preceding $\delta^*(\delta_k)$.

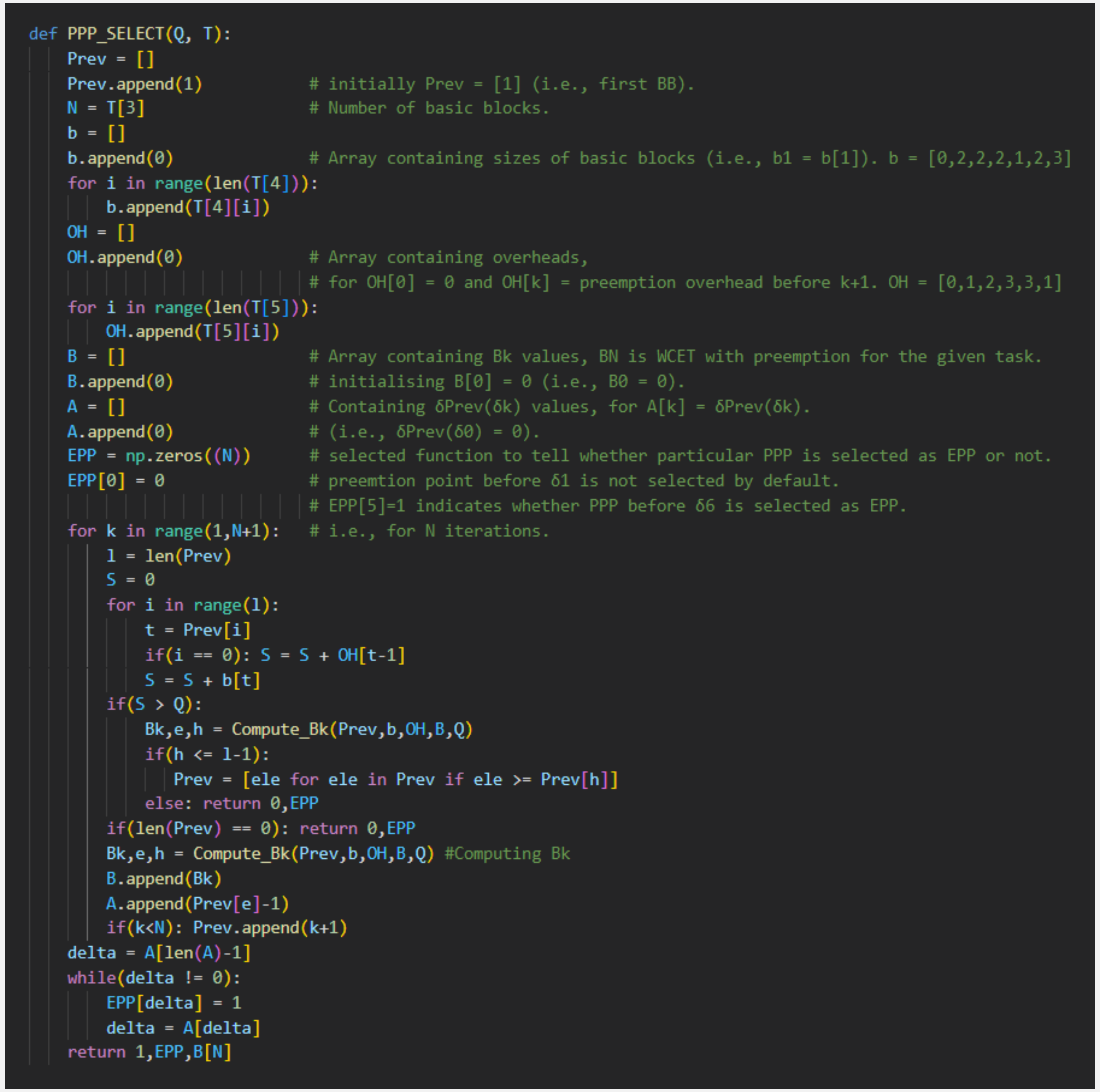
- Thus, the last EPP will be at the end of $\delta_{Prev}(\delta_N)$, penultimate EPP will be $\delta_{Prev}(\delta_{Prev}(\delta_N))$ and so on.

Algorithm



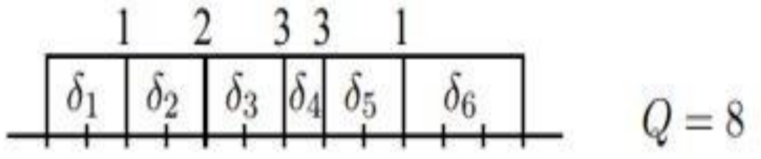
```
PPP_SELECT($Q, \tau$)
    Initialize: $Prev_1 \leftarrow \{\delta_1\}$, $B_0 \leftarrow 0$
1   for ($k$ : $1 \leq k \leq N$)
2           Remove from $Prev_k$ all $\delta_j$ violating (6)
3           if ($Prev_k = \emptyset$)
4                   return (Infeasible)
5           Compute $B_k$ using Equation (8)
6           Store $\delta_{Prev}(\delta_k)$
7           $Prev_{k+1} \leftarrow Prev_k \cup \{\delta_k\}$
    endfor
8   $\delta_j \leftarrow \delta_{Prev}(\delta_N)$
9   while ($\delta_j \neq \emptyset$)
10          Select the PPP at the end of $\delta_{Prev}(\delta_j)$
11          $\delta_j \leftarrow \delta_{Prev}(\delta_j)$
    endwhile
12  return (Feasible)
```

Fig. 3. Algorithm for the optimal selection of PPPs of a task.



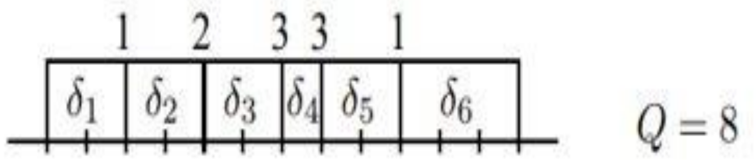
```python
def PPP_SELECT(Q, T):
    Prev = []
    Prev.append(1)              # initially Prev = [1] (i.e., first BB).
    N = T[3]                    # Number of basic blocks.
    b = []
    b.append(0)                 # Array containing sizes of basic blocks (i.e., b1 = b[1]). b = [0,2,2,2,1,2,3]
    for i in range(len(T[4])):
        b.append(T[4][i])
    OH = []
    OH.append(0)                # Array containing overheads,
                                # for OH[0] = 0 and OH[k] = preemption overhead before k+1. OH = [0,1,2,3,3,1]
    for i in range(len(T[5])):
        OH.append(T[5][i])
    B = []                      # Array containing Bk values, BN is WCET with preemption for the given task.
    B.append(0)                 # initialising B[0] = 0 (i.e., B0 = 0).
    A = []                      # Containing δPrev(δk) values, for A[k] = δPrev(δk).
    A.append(0)                 # (i.e., δPrev(δ0) = 0).
    EPP = np.zeros((N))         # selected function to tell whether particular PPP is selected as EPP or not.
    EPP[0] = 0                  # preemtion point before δ1 is not selected by default.
                                # EPP[5]=1 indicates whether PPP before δ6 is selected as EPP.
    for k in range(1,N+1):      # i.e., for N iterations.
        l = len(Prev)
        S = 0
        for i in range(l):
            t = Prev[i]
            if(i == 0): S = S + OH[t-1]
            S = S + b[t]
        if(S > Q):
            Bk,e,h = Compute_Bk(Prev,b,OH,B,Q)
            if(h <= l-1):
                Prev = [ele for ele in Prev if ele >= Prev[h]]
            else: return 0,EPP
        if(len(Prev) == 0): return 0,EPP
        Bk,e,h = Compute_Bk(Prev,b,OH,B,Q) #Computing Bk
        B.append(Bk)
        A.append(Prev[e]-1)
        if(k<N): Prev.append(k+1)
    delta = A[len(A)-1]
    while(delta != 0):
        EPP[delta] = 1
        delta = A[delta]
    return 1,EPP,B[N]
```

Example

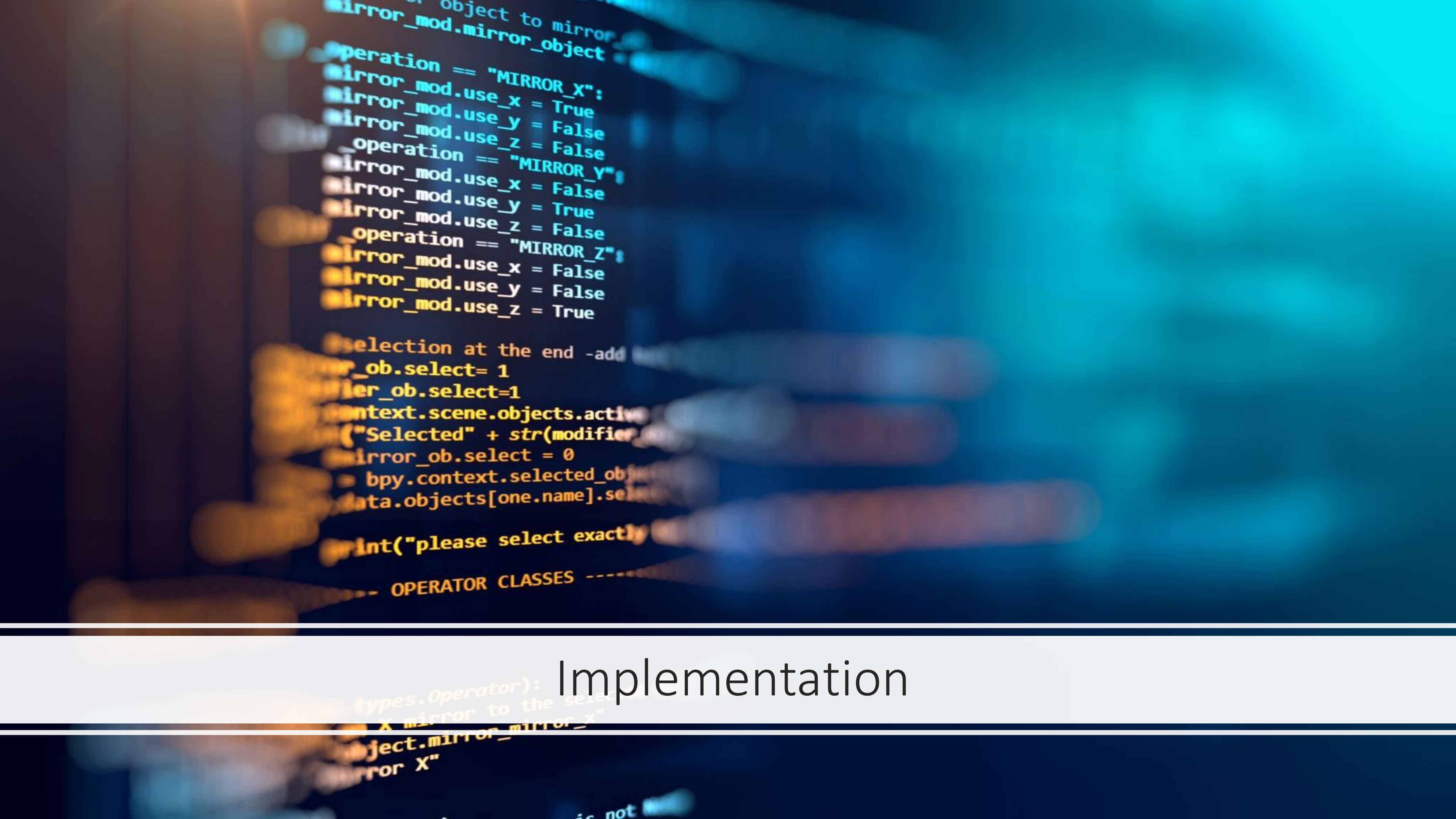


- $b_1 = 2$, $b_2 = 2$, $b_3 = 2$, $b_4 = 1$, $b_5 = 2$, $b_6 = 3$ (lengths of basic blocks)
- Initially $B_0 = 0$ and $Prev_1 = \{\delta_1\}$ (initialized)
- Then repeatedly checking the eq1 and eq2 for number of PPP times while updating, computing & storing $Prev_{next}$, B_k & $\delta_{Prev}(\delta_k)$ values respectively. We remove all the basic blocks that violate eq1 from $Prev_k$
- If the $Prev_k$ is empty while moving on to next PPP before all PPP's are covered the task is not feasible
- Now here until 1st 4 times in loop the eq1 doesn't violate is in Prev values every time. i.e., for k =4 $Prev_4 = \{\delta_1,\delta_2,\delta_3,\delta_4\}$ (eq1 -> $\xi_0 + b_1 + b_2 + b_3 + b_4 = 0+2+2+2+1 = 7<8$)
- So, until then δ_{Prev} (*) will be ϕ

Example



- When k = 5, $Prev_5 = \delta_1,\delta_2,\delta_3,\delta_4,\delta_5$ (eq1 -> $7+b_5 = 9 > Q$) so we remove δ1 from $Prev_5$ placing EPP after δ_1, $Prev_5 = \{\delta_2,\delta_3,\delta_4,\delta_5\}$now $B_5 = 10$ (from eq2) , $\delta_{Prev}(\delta_5) = \delta_1$. And now $Prev_6$ will be $\{\delta_2,\delta_3,\delta_4,\delta_5,\delta_6\}$
- Now at k = 6, (eq1 -> $\xi_1+b_2+b_3+b_4+b_5+b_6 = 1+2+2+1+2+3 = 11>Q$) so now 1st check eq by removing δ2 since it violates again check by removing δ_2 and δ_3 (eq1 -> $\xi_3+b_4+b_5+b_6 = 3+1+2+3 = 9>Q$) since it violates too now check again by removing δ_2, δ_3 and δ_4 (eq1 -> $\xi_4+b_5+b_6 = 3+2+3 = 8 = Q$) now it doesn't violate the condition so $Prev_6 = \delta_5$ and $\delta_{Prev}(\delta_6) = \delta_5$. $B_6 = B_5+\xi_3+b_6 = 14$ (from eq2)
- Since the end has been reached, B6 = 14 represents the minimum WCET. Thus, optimal WCET, Ci = B6 = 14 with EPP's after δ1 and δ5.



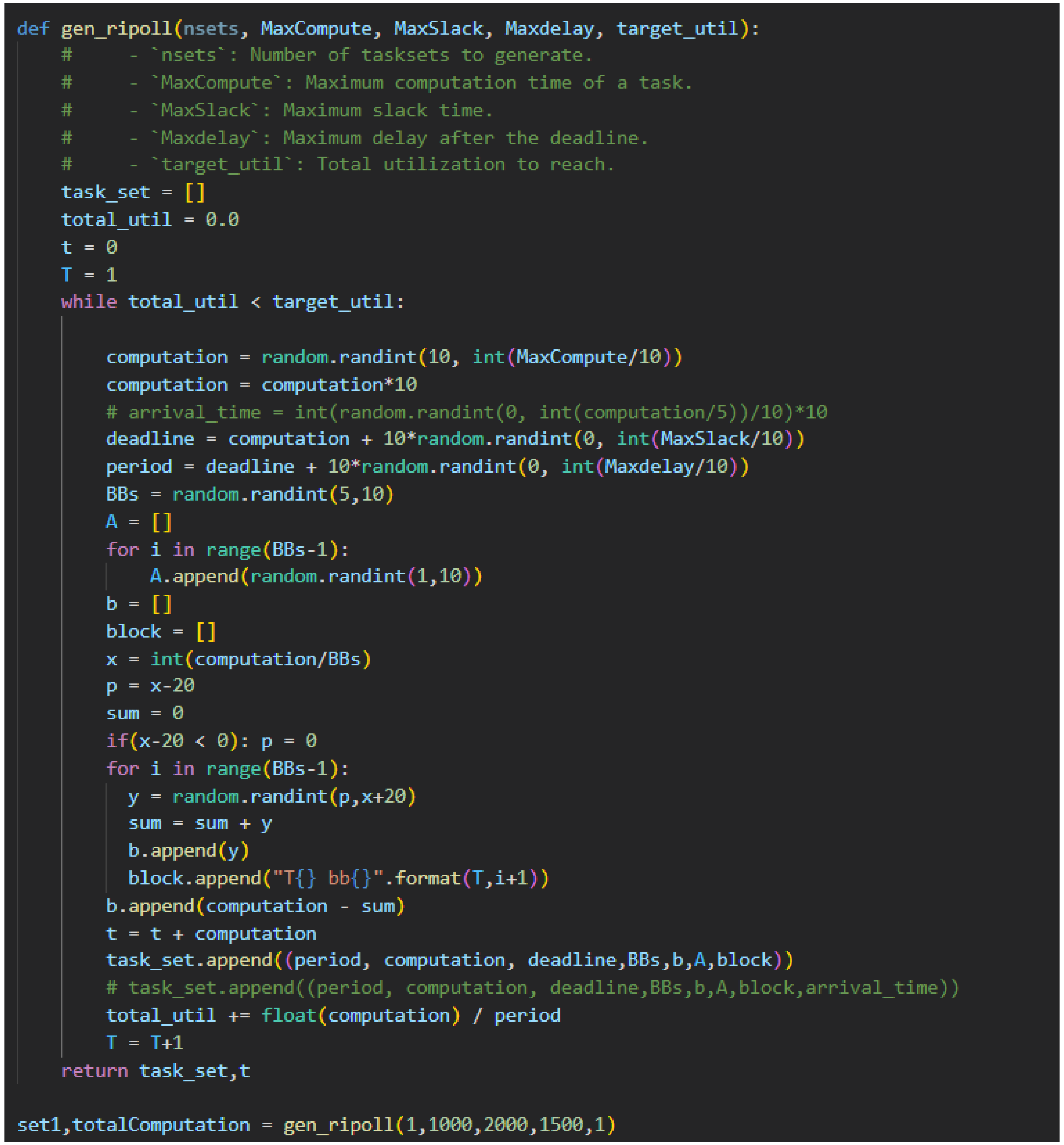
```
object to mirror
irror_mod.mirror_object
peration == "MIRROR_X":
irror_mod.use_x = True
irror_mod.use_y = False
irror_mod.use_z = False
_operation == "MIRROR_Y":
irror_mod.use_x = False
irror_mod.use_y = True
irror_mod.use_z = False
_operation == "MIRROR_Z":
irror_mod.use_x = False
irror_mod.use_y = False
irror_mod.use_z = True

election at the end -add
_ob.select= 1
er_ob.select=1
ntext.scene.objects.active
("Selected" + str(modifier
irror_ob.select = 0
bpy.context.selected_ob
ata.objects[one.name].se

int("please select exactly

- OPERATOR CLASSES ----
```

Implementation

Task Generation

- Task set is generated by modifying a task generation algo given by UUniFast algo.
- Computation time of each job is taken as a random number between 100 and a user given max computation time of a task in multiples of 10.
- Deadline is considered as computation time plus a random number between 0 and a user given max slack time of a task in multiples of 10.
- Period is considered as deadline plus a random number between 0 and a user given max delay after deadline of a task in multiples of 10.

Task Generation



```python
def gen_ripoll(nsets, MaxCompute, MaxSlack, Maxdelay, target_util):
    #     - `nsets`: Number of tasksets to generate.
    #     - `MaxCompute`: Maximum computation time of a task.
    #     - `MaxSlack`: Maximum slack time.
    #     - `Maxdelay`: Maximum delay after the deadline.
    #     - `target_util`: Total utilization to reach.
    task_set = []
    total_util = 0.0
    t = 0
    T = 1
    while total_util < target_util:

        computation = random.randint(10, int(MaxCompute/10))
        computation = computation*10
        # arrival_time = int(random.randint(0, int(computation/5))/10)*10
        deadline = computation + 10*random.randint(0, int(MaxSlack/10))
        period = deadline + 10*random.randint(0, int(Maxdelay/10))
        BBs = random.randint(5,10)
        A = []
        for i in range(BBs-1):
            A.append(random.randint(1,10))
        b = []
        block = []
        x = int(computation/BBs)
        p = x-20
        sum = 0
        if(x-20 < 0): p = 0
        for i in range(BBs-1):
          y = random.randint(p,x+20)
          sum = sum + y
          b.append(y)
          block.append("T{} bb{}".format(T,i+1))
        b.append(computation - sum)
        t = t + computation
        task_set.append((period, computation, deadline,BBs,b,A,block))
        # task_set.append((period, computation, deadline,BBs,b,A,block,arrival_time))
        total_util += float(computation) / period
        T = T+1
    return task_set,t

set1,totalComputation = gen_ripoll(1,1000,2000,1500,1)
```

Computation of Q

- We computed Q such that there is atleast one basic block present in non - pre-emptive region.
- In the paper Q is computed by considering feasibilty into account for EDF and FP.
- This is done by schedulability analysis which resulted in below expressions

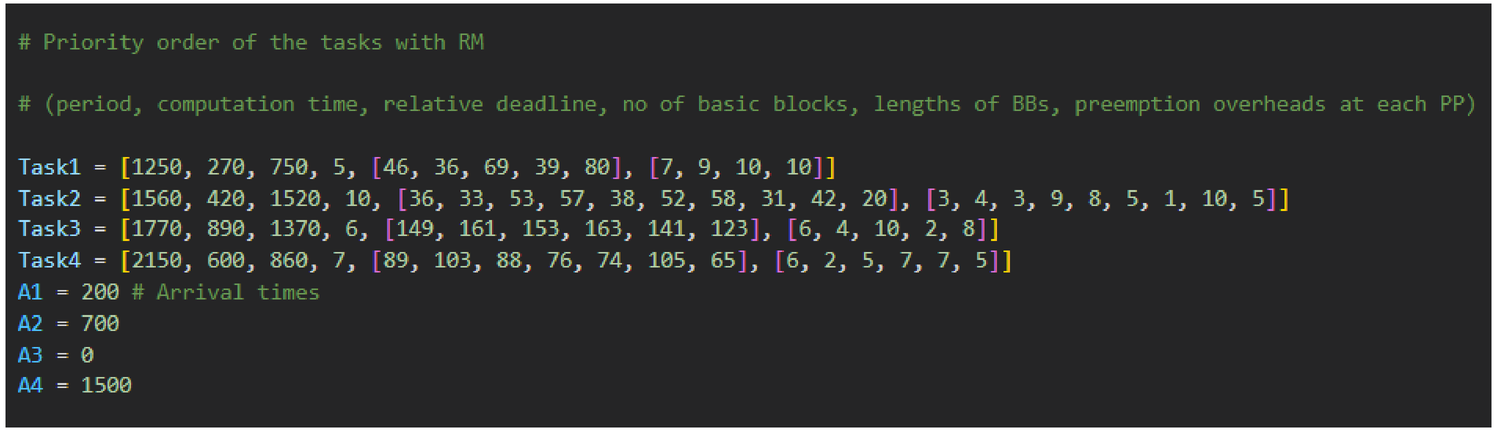
$$\beta_i^{\mathrm{EDF}} \doteq \min_{a \in A | D_i \le a < D_{i+1}} \left\{ a - \sum_{\tau_j \in \tau} \mathrm{DBF}_j(a) \right\},$$

$$\mathrm{DBF}_i(a) = \left(1 + \left\lfloor \frac{a - D_i}{T_i} \right\rfloor\right) C_i.$$

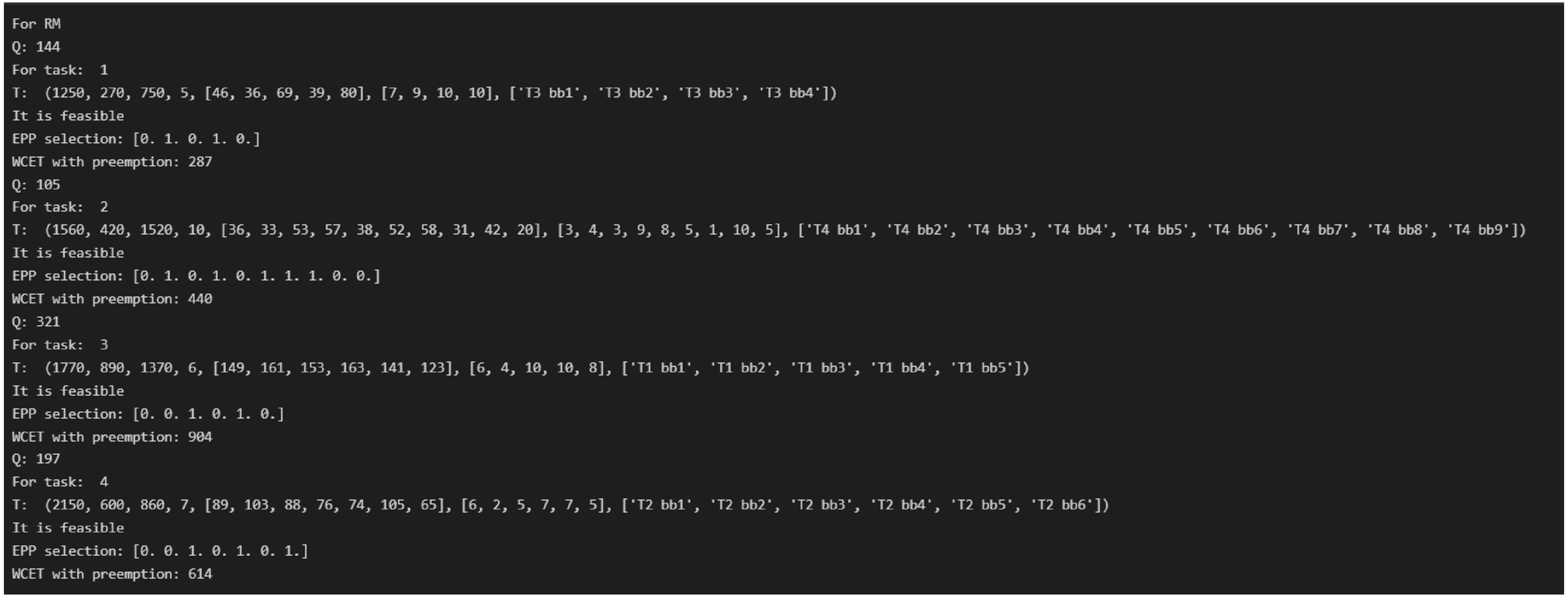
with $A = \{kT_j + D_j,\ k \in \mathbb{N},\ 1 \le j \le n\}.$

- But with our implementation, for most of the tasks generated, set for value "a" was empty which is resulting in no update of Q.

Basic Example:-

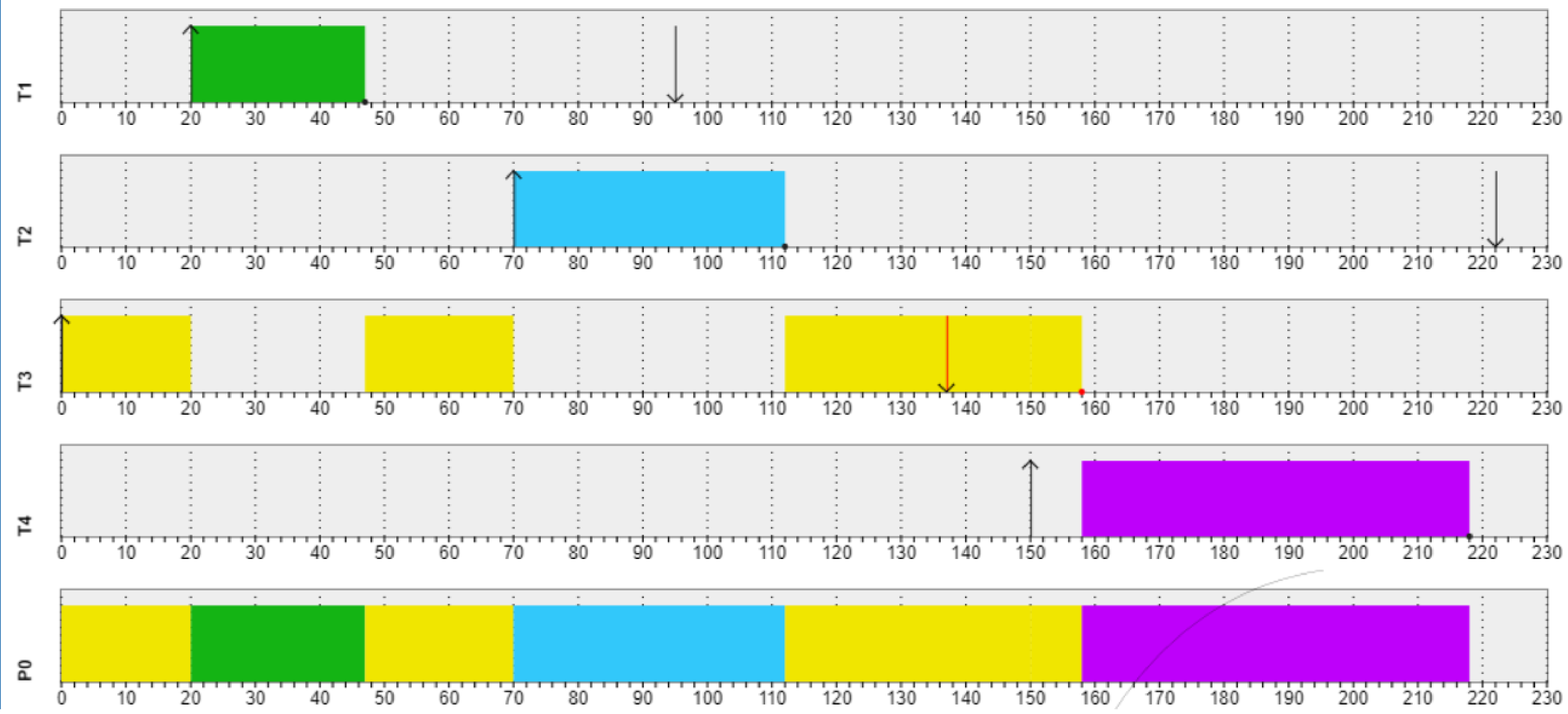


```
# Priority order of the tasks with RM

# (period, computation time, relative deadline, no of basic blocks, lengths of BBs, preemption overheads at each PP)

Task1 = [1250, 270, 750, 5, [46, 36, 69, 39, 80], [7, 9, 10, 10]]
Task2 = [1560, 420, 1520, 10, [36, 33, 53, 57, 38, 52, 58, 31, 42, 20], [3, 4, 3, 9, 8, 5, 1, 10, 5]]
Task3 = [1770, 890, 1370, 6, [149, 161, 153, 163, 141, 123], [6, 4, 10, 2, 8]]
Task4 = [2150, 600, 860, 7, [89, 103, 88, 76, 74, 105, 65], [6, 2, 5, 7, 7, 5]]
A1 = 200 # Arrival times
A2 = 700
A3 = 0
A4 = 1500
```

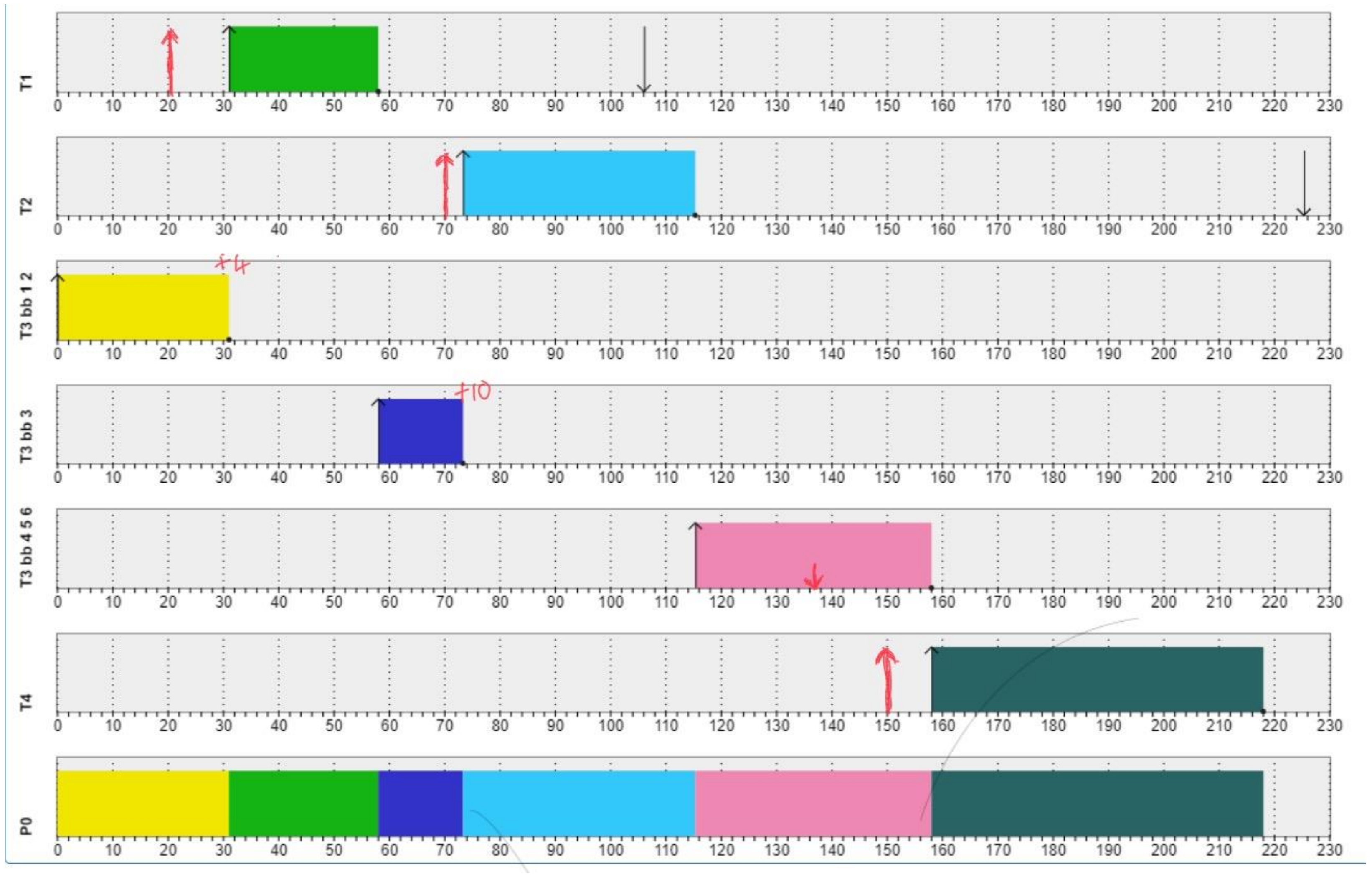


```
For RM
Q: 144
For task:  1
T:  (1250, 270, 750, 5, [46, 36, 69, 39, 80], [7, 9, 10, 10], ['T3 bb1', 'T3 bb2', 'T3 bb3', 'T3 bb4'])
It is feasible
EPP selection: [0. 1. 0. 1. 0.]
WCET with preemption: 287
Q: 105
For task:  2
T:  (1560, 420, 1520, 10, [36, 33, 53, 57, 38, 52, 58, 31, 42, 20], [3, 4, 3, 9, 8, 5, 1, 10, 5], ['T4 bb1', 'T4 bb2', 'T4 bb3', 'T4 bb4', 'T4 bb5', 'T4 bb6', 'T4 bb7', 'T4 bb8', 'T4 bb9'])
It is feasible
EPP selection: [0. 1. 0. 1. 0. 1. 1. 1. 0. 0.]
WCET with preemption: 440
Q: 321
For task:  3
T:  (1770, 890, 1370, 6, [149, 161, 153, 163, 141, 123], [6, 4, 10, 10, 8], ['T1 bb1', 'T1 bb2', 'T1 bb3', 'T1 bb4', 'T1 bb5'])
It is feasible
EPP selection: [0. 0. 1. 0. 1. 0.]
WCET with preemption: 904
Q: 197
For task:  4
T:  (2150, 600, 860, 7, [89, 103, 88, 76, 74, 105, 65], [6, 2, 5, 7, 7, 5], ['T2 bb1', 'T2 bb2', 'T2 bb3', 'T2 bb4', 'T2 bb5', 'T2 bb6'])
It is feasible
EPP selection: [0. 0. 1. 0. 1. 0. 1.]
WCET with preemption: 614
```

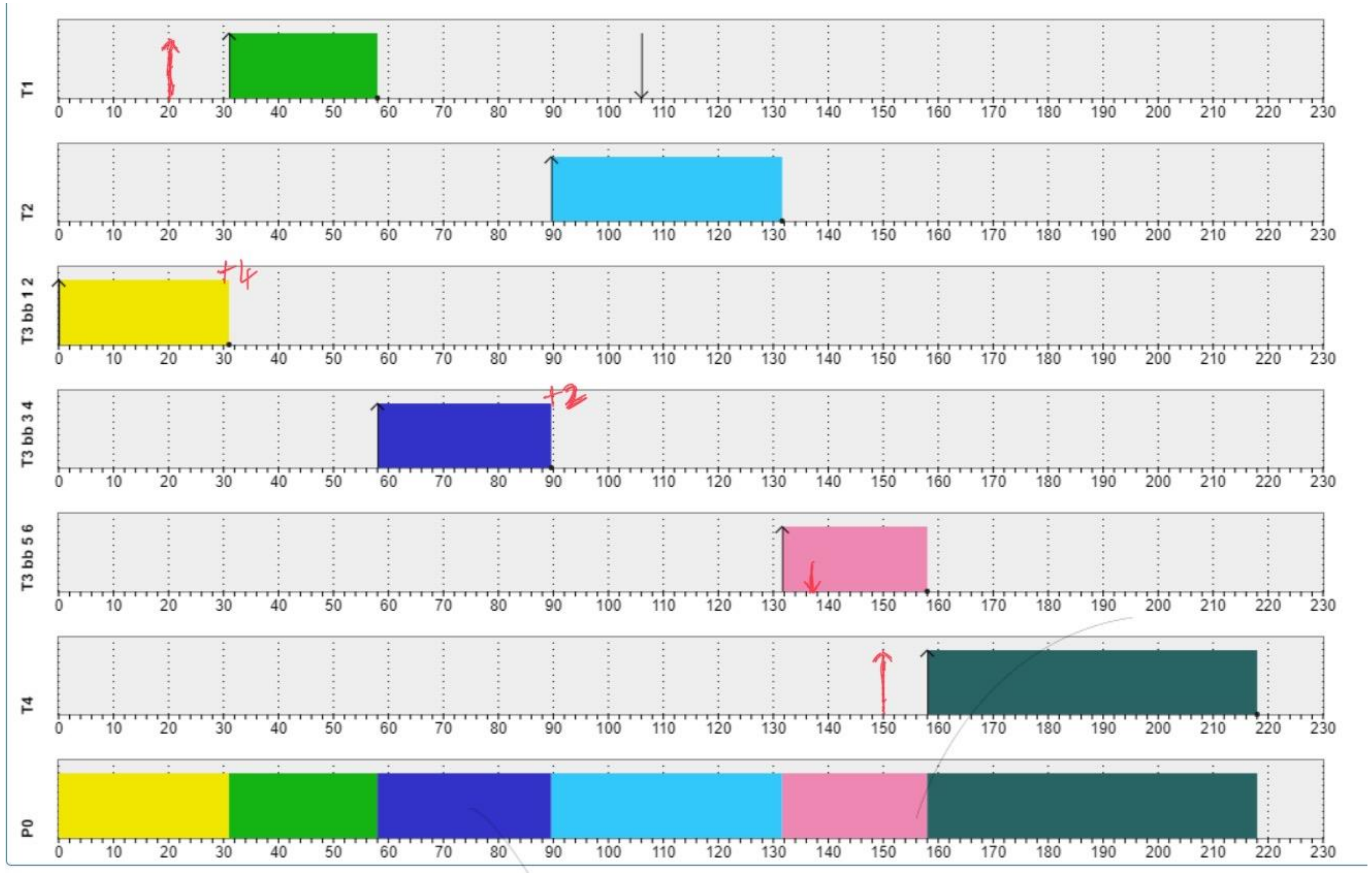
Basic Example
Fully Preemptive Without considering BB's & OH's



Basic Example :
Preemptive at all PPP's

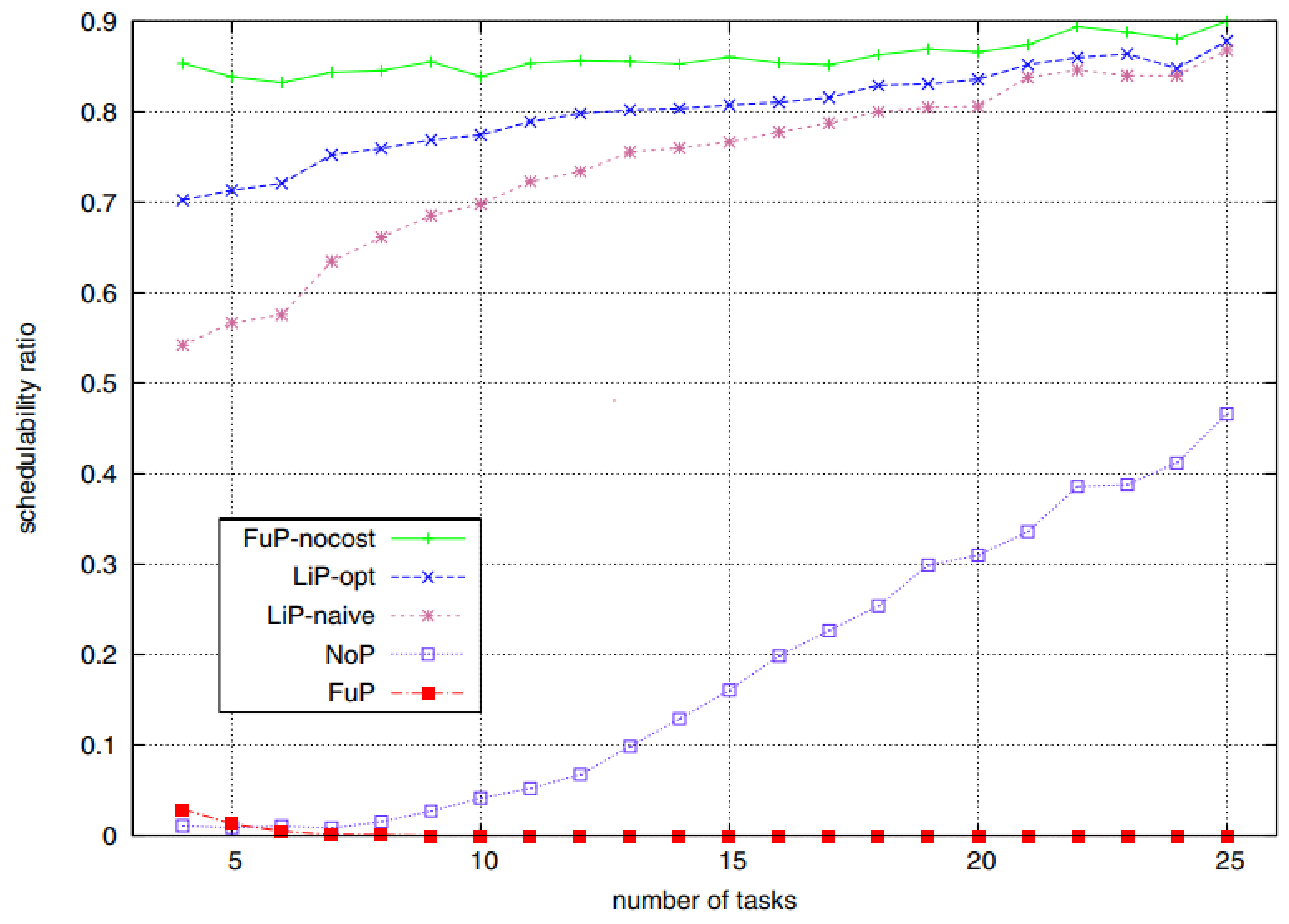


Basic Example
Preemptive at all EPP's (gives less overhead, CRPD)

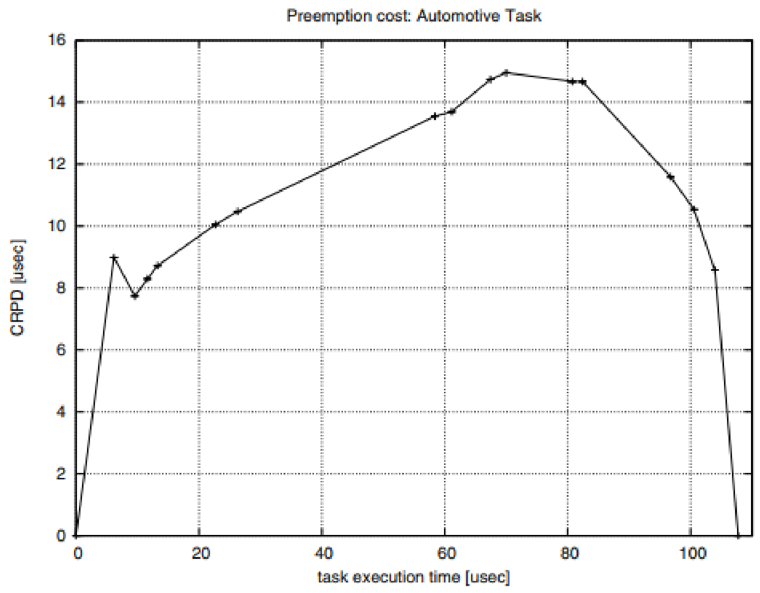


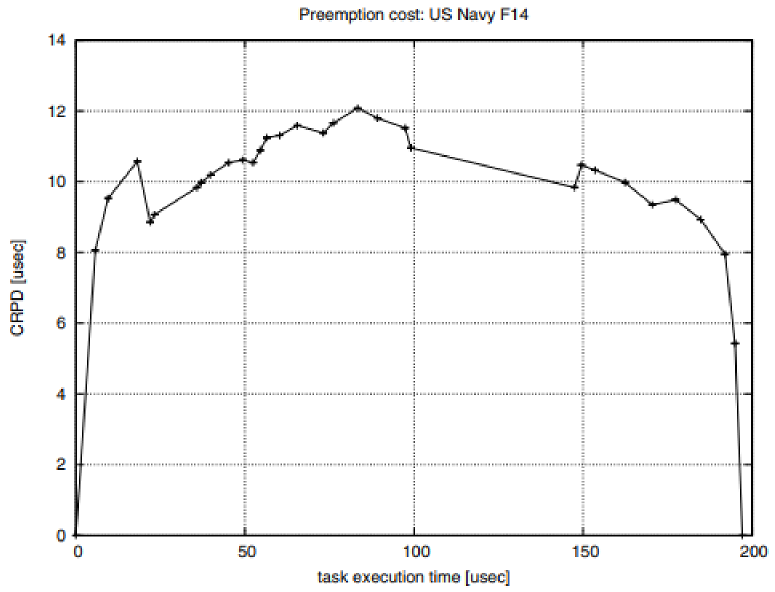
Theoretical Comparision

- **FuP-nocost**: Fully preemptive scheduler with no Overhead
- **FuP**: Fully preemptive scheduler with preemption cost
- **LiP-naive**: Limited preemption approach with variable cost
- **NoP**: Non-preemptive algorithm
- **LiP-opt**: Algo proposed in paper

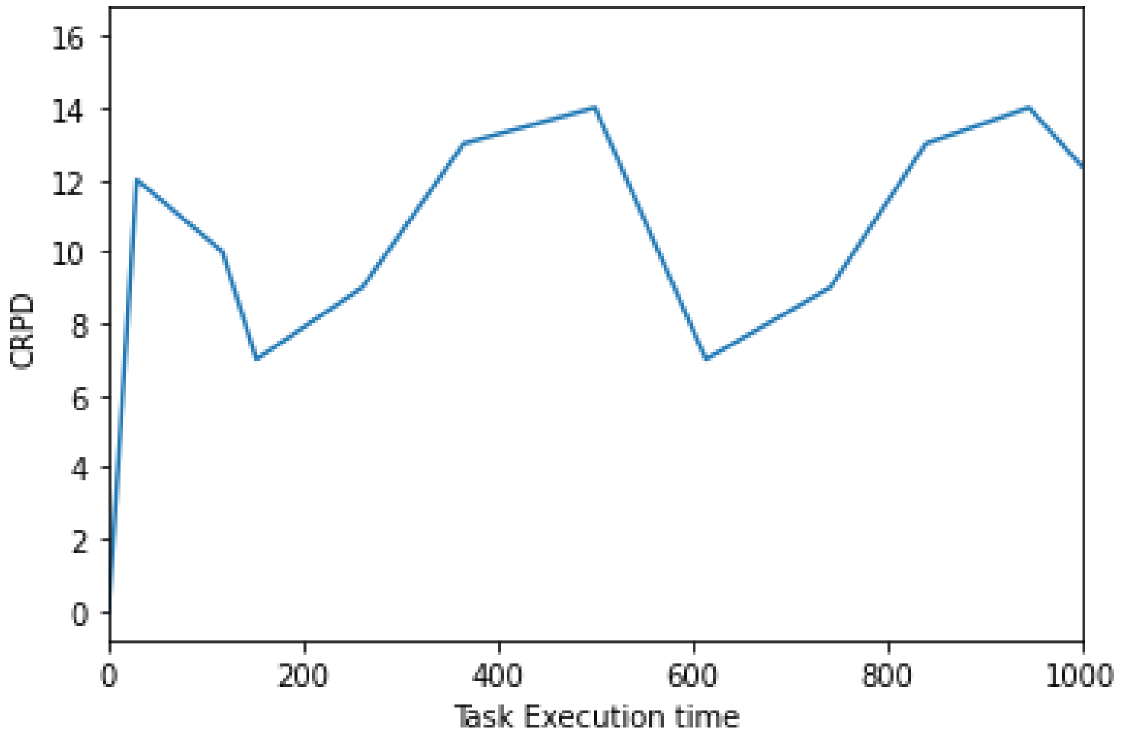


Preemption cost:





- The x-axis represents the time at which a preemption takes place, while the y-axis represents the increase in the execution time caused by a preemption at that place. Where OH is randomized between 5 & 16.



Advantages

- Our method relaxes assumption of taking equal preemption costs throughout the code, accounting for more realistic scenario.
- Smart preemption point selection can significantly reduce the WCET, cache related delays and context switch time of each task.
- Finds much more efficient solution that has a linear complexity both in space and time. (Assumptions considered)
- If a feasible schedule is not found by the proposed method, then no other strategy can lead to a feasible solution.(Assumptions considered)

Critiques

Expressing the preemption overhead as a function of the preempting task would significantly complicate the analysis and to avoid this, following assumptions are considered

- A1:- **The cache is cold after each context switch** (fixed overhead)
- A2:- **Each EPP leads to a preemption**

However, the above assumptions are pessimistic.

A1: Preempting task may have small footprint than the overhead resulted by this assumption.

A2: There might be cases in which not all EPPs leads to preemption.

The optimality of the proposed algorithm for the activation of the preemption points completely depends on assumptions A1. and A2

Thank you