SlotShifting to CapacityShifting

L.J. GokulVasan
Chair of Real-Time Systems
TUKL, Germany
Email: lakshama@rhrk.uni-kl.de

Dr.Mitra Nasri
Chair of Real-Time Systems
TU-KL, Germany
Email: nasri@eit.uni-kl.de

Prof.Dipl.-Ing.Dr.Gerhard Fohler
Chair of Real-Time Systems
TU-KL, Germany
Email: fohler@eit.uni-kl.de

Abstract

SlotShifting algorithm provides a mechanism to accommodate Event triggered task along with periodic tasks without explicit bandwidth allocation. Though algorithm has a very beautiful approach to solve the problem of admittance it has its own disadvantages and overheads (mentioned in section 2 of the paper)which is been addressed in the following paper. On addressing the problems the important notion of slot shifting,i.e. slots(TDMA) is eradicated , thus leading to new algorithmic approach called Capacity Shifting. We will show that this approach in major cases reduces the scheduling overhead by 60 percent.

In this paper we will also derive a new Guarantee algorithm which is much efficient and faster than existing one.

I. Introduction

Joint scheduling of periodic and aperiodic is made possible in Slotshifting by computing guarantee for the periodic tasks (whose earliest start time is known deterministically advance)in offline and used in online phase to decide on admittance of aperiodic task.The beginning of this section will overtone on some parts of the slotshifting algorithm which would succor in establishing the problem statement.

The online selection function in slotshifting is EDF and decision function is triggered at beginning of every slot.since decision functionality of the system is based on slot the computation time of the tasks are also computed and rounded to slots,.i.e. worst case execution time of a task is measured based number of slots metrics rather than continuous time metrics. To achieve guarantee for tasks a layer of certainty is applied around the EDF algorithm called interval.Jobs of tasks gets associated with interval based on its deadline.In offline phase The periodic taskset is tested for schedulability and table of intervals are generated for periodic tasks for a hyper period. The interval holds an additional field which is of our interest called Spare capacity.In online phase the Amount of availability of spare capacity from the current time till the deadline of the aperiodic task is used for deciding acceptance of aperiodic task.The availability of spare capacity for each interval is calculated using formulae.[1]

$$SC_{I_i} = |I_i| - \sum_{i=0}^{n} C_i + \min(0, SC_{I_{i+1}}) \tag{1}$$

in words, Spare-Capacity of an interval is the length of the interval negated from summation of computation time needed by the jobs that belongs to the interval added with the minimum of 0 and spare capacity of the next interval. This creates a wave of negative spare capacity backwards. The same formulae is applied both in offline and during online phase when accepted aperiodic task creates new interval.

During online phase as time progresses the interval table also progresses forward, The interval that is associated with the current time is called current interval.For each slot the job selected based on EDF is checked for interval to which it belongs, if task belongs to current interval then the spare capacity of the interval is untouched because the needed computation is already negated in offline, else spare capacity of the current interval is negated by one slot and spare capacity of the job's interval is tested for negativity, if negative then along with job's interval all the intervals that are negative backwards are added one slot to its spare capacity till it reaches a positive interval, else just the job's interval is added with one slot.This whole process repeats itself for every slot, This whole process that is performed every slot is called update spare capacity.We name this whole functionality of updating spare capacity every slot as **update spare capacity or update-sc**

Though Slotshifting has a beautiful approach to accept and guarantee Event based jobs without explicit bandwidth allocation, The computation that needs to be done during online phase supersedes its advantages.The next section provides some of disadvantages which I tried to address in this paper.

[1]The notations are same as [FH94], we will later change the notation into a much easier single letter representation

II. Problem Statement

• ***Problem1:*** *Updating Spare Capacity*

the function update-sc is applied for every slot.The problem arises when job that does not belong to the current interval is executing in current interval and spare capacity of the job's Interval is negative then we traverse Backwards through all the consecutive negative intervals and increment the spare capacity of the intervals till either we reach positive interval or current interval, which we presume is the lender to these backward consecutive intervals.

The function update-sc becomes an consistent Overhead when

- current executing job's Interval is not Current interval and job's interval's spare capacity is a BIG negative number, which intuitively means the borrow from the lender interval is also big.
- Interval to which task belongs is far away from lender,i.e. there are many intervals in between the lender and job's interval.
- overhead further complicates when current job is the only task being selected for the next consecutive slots of Current Interval, which might be a normal scenario in EDF.

• ***Problem2:****Slots*

Notion of slots might be a good approach in distributed systems but the system that has centralised clock trigger, slot adds following overhead.

- Increases the number of time scheduling decision needs to be made.
- The worst case execution time of the task is calculated based on[1]:

$$MAX(T) = \lceil \frac{calculateduppperboundofT'sexecutiontime}{slotlength} \rceil \tag{2}$$

In the above defined equation, We round up the computation time of the tasks to fit the notion of slot,In some cases the mentioned computation takes off more time from system than needed by the tasks and could fail admittance of some aperiodic tasks.

• ***Problem3:*** *Acceptance and Guarantee Algorithm*

In online phase, when an aperiodic arrives and gets accepted then it needs to be guaranteed. Guarantee Algorithm in short creates a new interval if needed and starts recalculating the spare capacity from the last interval in which the deadline of the aperiodic falls till the current interval applying the formulae 1 described previously. The traditional spare capacity calculation might be helpful in offline phase for initial spare capacity calculation; but in online phase this calculation adds following complications

- After an aperiodic is accepted, In order to Guarantee the accepted task, we need to recalculate the spare capacity from the interval in which aperiodic was accepted till the current interval. Along with the backward consecutive traversal through each interval till current interval; Guaranteeing needs traversal through the jobs of each interval till the current interval.The job traversal in each interval adds following complexities.
  - both job and interval needs to hold reference of each other, needing an additional place holder for holding reference.
  - Adds additional difficulty in implementation.
  - Increases runtime complexity in calculation of spare capacity for each interval.
  - Calculation of spare capacity on each interval would take a different runtime which directly proportionates the number of jobs associated with the interval,in other words complexity of calculation of spare capacity per interval is not constant, reducing the determinism of the system.

III. System Model and Background

Task set. Consider a real-time system with n periodic tasks, $\tau_1 ... \tau n$. Each task τ_i has a worst case computation requirement C_i. A period T_i, an initiation time or offset relative to some time origin Φ, where $0 \leq \phi_i \leq T_i$ and a hard deadline(relative to the initiation time)d_i. The parameters C_i, T_i, ϕ_i, d_i are assumed to be known deterministic quantities. We require that the tasks be scheduled according to Earliest deadline first scheduling algorithm with τ_1 having highest priority 1 and τ_n having lowest priority; however we do not require this priority assignment to hold, but we do assume that $d_i \leq T_i$.

A periodic task, say τ_i, gives rise to an infinite sequence of jobs. The k^{th} such job , $J_{i,k}$ is ready at time $R_{J_{i,k}} = \phi_i + (K-1)T_i$ and its $C_{J_{i,k}}$ units of required execution must be completed by time $d_{J_{i,k}} = \phi_i + (K-1)T_i + d_i$ or else *timing fault* will occur causing the job to terminate.We assume a fully preemptive system.

We now introduce the aperiodic tasks, $\gamma_k, K \geq 1$. Each aperiodic job, γ_k, has an associated arrival time, α_k, a processing requirement, p_k, and a hard deadline, D_k. The tasks that arrives with such hard deadline is called firm

[1]The notations are same as [FH94], we will later change the notation into a much easier single letter representation

aperiodic tasks,Γ and tasks with no hard deadline is named soft aperiodic tasks, ζ, if the aperiodic task does not have a hard deadline, we set $D_k = None$. The aperiodic tasks are indexed such that $0 \leq \alpha_k \leq \alpha_{k+1}, K \geq 1$. we assume that the aperiodic task sequence is not known in advance.The list holding set of firm aperiodic task waiting to be accepted is called unresolved list,ι_A. The list holding set of tasks which could not be guaranteed and ζ is called not guaranteed list,ι_S, job from such a list is named $J_{i,k,S}$. Irrespective of the list, Job that is selected and running in the system is called current job, $J_{i,k,curr}$.

offline phase. The definition of interval is as follows, The End of interval $e_i = d_{J_{i,k}}$, The end of an interval determines the ownership of the jobs that belong to the interval and there could be more than one job associated with an interval. The early start time of an interval $\xi_i = \min(R_{J_{i,k}})$,i.e. is the minimum of the start time of the job that belongs to the interval. The start of an interval $S_i = \max(e_{i-1}, \xi_i)$. The spare capacity of an interval ω_i for an interval is defined by $\omega_i = |\omega_i| - \sum_{i=0}^{n} C_{J_{i,k}} + \min(0, \omega_{i+1})$. Spare capacity and interval calculation notion could create a negative spare capacity interval which could create a backward consecutive wave of negative intervals till a interval which could completely satisfy the capacity needed for such interval, This ripple due to negative spare capacity backward propagation forms a relation among these intervals which we name as *relation window*,ϱ_i,The first interval of this relation window is named *lender*,l_i and last in the relation window is named *lent-till*, b_i. There could be many such relation window in a system within its hyper period. The interval progression is closely associated with time, the interval that is assocated with the current time is called current interval, I_{curr}. The interval to which job $J_{i,k}$ is associated is named $I_{J_{i,k}}$. The list or table of intervals is simply named I.

Online Phase. The **timer** τ is triggered based on the parameter τ_{expiry}, which in case of slot shifting is a fixed relative period called slots. A **slot** s is defined as an external observer (τ) counts the ticks of the globally synchronized clock with granularity of slot length,$|s|$, i.e. $\tau_{expiry} \leftarrow |s|$ and assigns numbers from 0 to ∞ on every slot called slot count s_{cnt}. We denote by "in slot i" the time between the start and end of slot i, s_i i.e. the time-interval $[|s| * i, |s| * (i + 1)]$. slot has uniform time length and on expiry triggers function schedule.

Now I sketch the existing version of slot shifting algorithm which would give us a deep intuitiveness of the problem, thus easing the justification of the proposed one.

Algorithm 1: Slotshifting Online Phase

```
1  Function schedule(J_{i,k,Curr});
   Input  : J_{i,k,Curr}: job previously selected
   Output: J_{i,k,nxt}: next selected job
2  update-sc(J_{i,k,Curr}, I)                          ▷ update the interval;
3  if A != None then
4  |  Test-aperiodic(A)                  ▷ Test and try Guarantee Aperiodic's if any;
5  end
6  J_{i,k,nxt} ←selection-fn()                         ▷ Get next eligible job;
7  update-slot();
8  return J_{i,k,nxt};

9  Function update-sc(J_{i,k,Curr}, I);
   Input  : J_{i,k,Curr}: previous selected job, I: Interval list
   Output: None
   Result: Spare Capacity of the respective intervals get updated
10 if J_{i,k,Curr}.I == I_{curr} then
11 |  return;
12 end
13 I_{curr} ← I_{curr} − 1;
14 I_{tmp} ← J_{i,k,Curr}.I;
15 while I_{tmp} do
16 |  if I_{tmp}.ω ≥ 0 then
17 |  |  I_{tmp}.ω ← I_{tmp}.ω + 1;
18 |  |  break;
19 |  end
20 |  else
21 |  |  I_{tmp}.ω ← I_{tmp}.ω + 1;
22 |  |  I_{tmp}.ω ← I_{tmp}.prev;
23 |  end
24 end
```

```
1  Function update-slot();
   Input  : None
   Output: None
   Result: s_cnt gets added by 1; also check and move I_curr to I.nxt
2  s_cnt ← s_cnt + 1;
3  if s_cnt ≥ I_curr.end then
4  |  I_curr ← I_curr.nxt;
5  end

6  Function Test-aperiodic(ι_A, I);
   Input  : ι_A: Unconcluded firm aperiodic list;
            I: list of intervals
   Output: None
   Result: Job in ι_A is moved to either ι_G or ι_S
7  while ι_A != None do
8  |  Γ ← A.dequeue();
9  |  if I_acc ← Acceptance-test(Γ, I) != None then
10 |  |  Guarantee-job(I, I_acc, Γ);
11 |  |  ι_G.queue(Γ)
12 |  end
13 |  else
14 |  |  ι_S.queue(Γ)
15 |  end
16 end

17 Function Acceptance-test(Γ, I);
   Input  : Γ: Unconcluded firm aperiodic job;
            I: list of intervals
   Output: I_acc: interval in which task was accepted or None
18 I_tmp ← I_curr;
19 ω_tmp ← 0;
20 while I_tmp.end ≤ Γ.d do
21 |  if I_tmp.ω > 0 then
22 |  |  ω_tmp ← I_tmp.ω + ω_tmp;
23 |  end
24 |  I_tmp ← I_tmp.nxt;
25 end
26 if  ω_tmp ≥ Γ.C then
27 |  return I_tmp.prev;
28 end
29 else
30 |  return None;
31 end
```

```
Function Guarantee-Job(I, I_acc, Γ);
  Input  : Γ: Unconcluded firm aperiodic job;
           I: list of intervals;
           I_acc: interval at which Γ.d falls
  Output: None
  Result: Γ gets guaranteed
if I_acc.end > Γ.C then
  | I_New = split-intr(I_acc, Γ.C);
end
 I_iter ← I_acc;
while I_iter.id ≥ I_curr.id do
  | J_lst ← I_iter.J_lst;
  | ω ← 0;
  | J ← J_lst.head;
  | while J do
  | | ω ← ω + J.c;
  | | J ← J.nxt;
  | end
  | I_iter.ω ← ω + max(0, I_iter.nxt.ω);
  | I_iter ← I_iter.prev;
end
```

IV. Proposed Solution: Capacity Shifting

In this section we define a new algorithm inspired from slotshifting but with the mentioned problems addressed hence moving towards a new flavour of algorithm called Capacity Shifting. To understand the solution we first need analyse the current algorithm and understand the intuitiveness behind the existing algorithm, making side by side comparison much justifiable. To make this approach easier we from here on call the existing algorithm *traditional* so that we could seamlessly distinguish between the existing algorithm and proposed one.

A. Deferred Update of Spare Capacity.

When the procedure **update-sc** in *Section 2 Algorithm 1* is noticed; If a job that does not belong to current interval is executed in the current interval and $I_{j,k}.\omega$ is negative then the function iterates backwards adding every interval with one additional slot till a positive interval or current interval,I_{curr} is found. Updating till I_{curr} is named *neutralisation effect*, because spare capacity is taken and given back. This process once starts probability of repeating in the next consecutive execution slots is high atleast till the $I_{j,k}.\omega$ becomes positive; causing unnecessary overhead. In worst case the complexity of update-sc is $O(n)$, where n is the number of consecutive backward interval traversal to update spare capacity.

This mentioned problem could be avoided by accumulating and updating the spare capacity at the end of the current interval and beginning of the next interval within ϱ_i or when needed scenarios like firm aperiodic arrival. This approach is named as **deferred update**.

Offline and preparation phase. To make this deferred update work, offline phase is complicated with additional computation which creates relation window for a set of intervals which has a relation of lender l_i and lent-till b_i. Every Interval is provided with three additional fields namely lender l, lent-till b and update-val u.During offline computation if the relation ϱ_i exists among set of intervals then the fields l and b are updated with the corresponding references.This is made use in the online phase through following algorithm which I call **O1-Update-SC**.

Algorithm 1: Slot based O1-Update-SC

Function o1-update-sc($J_{i,k,curr}, I$);
Input : $J_{i,k,curr}$: Previous selected job, I: Interval table
Output: None
Result: Checks and Updates only spare capacity of job and current interval

if $J_{i,k,curr}.interval == I_{curr}$ **then**
  return;
end
$tsk - \omega \leftarrow J_{i,k,curr}.interval.\omega$ ▷ take a local copy of task's spare capacity before updating;
if $J_{i,k,curr}.interval.\omega < 0$ **then**
  $J_{i,k,curr}.interval.u \; += 1;$
end
$J_{i,k,curr}.interval.\omega \; += 1;$

/***neutralisation effect** 1.check job's lender and current interval's lender is same.;
2. also check if job's interval is negative before update.;
if both condition satisfies just don't negate I_{curr};
*/;

if $J_{i,k,curr}.interval.l == I_{curr}.l \quad and \quad tsk - \omega < 0$ **then**
  return;
end
$I_{curr}.\omega \; -= 1;$

Described O1-Update-SC will get triggered at the end or beginning of every slot. when noticed the textit algorithm 1 section 4algorithm has no loops and just checks on couple of conditions to update spare capacity of job's interval or current interval's spare capacity:

- check spare capacity of job's interval,ω_j is negative before updating, if yes, then along with ω_j update also update job's interval's update-val, U_j.
- check lender of job's interval,l_j and lender of current interval, l_{curr} is same and also check ω_j is negative before updating. If both these conditions are satisfied then don't negate the ω_{curr}.

At any given scenario now the complexity of spare capacity update per slot is $O(1)$, but still the notion of slot exist in algorithm 2, which would be removed as we start addressing the problems in phases.I shall simply call Algorithm 2 as phase1 of moving towards slotless notion.

The above mentioned algorithm O1-Update-SC is just a facilitator of deferred update algorithm. The real spare capacity update for intervals happen during deferred update. The function deferred update gets triggered on 2 conditions.

- when I_{Curr} is moving to next interval
- when an aperiodic task arrives and needs to be tested for guarantee.

To understand the intuitiveness of the deferred update, I will now sketch an non working algorithm which I would call **naive-deferred-update** and explain certain challenges or scenarios that would defy straight forward notion of deferring the update of intervals within ϱ_i. I presume this approach would provide a good insight on the complete solution on deferred update.

Algorithm 2: naive-deferred-update

Function naive-deferred-update(I);
Input : I: Interval list
Result: spare capacity of the interval within ϱ_i is updated with right data

$lentTill \leftarrow I_{curr}.b$;
$updateVal \leftarrow 0$;
while $lentTill \; != I_{curr}$ **do**
  $lentTill.\omega \leftarrow lentTill.\omega + updateVal$ ▷ Update the interval with previous interval's deferred spare capacity;
  $updateVal \leftarrow updateVal + lentTill.U$ ▷ Accumulate previous intervals within ϱ_i update-val ;
  $lentTill.U \leftarrow 0$;
  $lentTill \leftarrow lentTill.prev$;
end

To brief on the Algorithm 2 mentioned above, The algorithm traverses backwards starting from lent-till,b_i till I_{curr}. As it moves backwards it does the following

- It starts accumulating intervals update-val, U_i, within ϱ_i generated during *O1-Update-SC*
- As it progresses backwards and before accumulating the current lent to interval's update-val. U_i, we add the previously accumulated U_i to the current lent to interval's spare capacity $b_i.\omega$.

To make the explanation easier, we will symbolize certain functions. The function Update-SC symbolized as $I_{J_{i,k}} \twoheadrightarrow I_{curr}$, where I_{curr} is the current interval that is executing a job that belongs to the interval $I_{J_{i,k}}$ and the function Update-SC is applied. We will symbolize the result as ω_i, ω_i represents the spare capacity of the interval i, The subscript,i that represent interval is made optional, rather the examples or scenario explanation holds the column representing the interval id. symbolizing the function O1-Update-SC as $I_{J_{i,k}} \twoheadrightarrow^{①} I_{curr}$, The super script ① represents $O(1)$ version of Update-SC, i.e. O1-Update-SC. The result of the $\twoheadrightarrow^{①}$ will be represented $\omega_i^{[U]}$, where superscript U is the update val of the interval i. We say $\twoheadrightarrow_{curr} I_i$ (No entity on the left), when the current interval moves from I_{curr} to I_i, .i.e. I_i becomes I_{curr}. We will also symbolize function deferred-update as $\curvearrowright \varrho_i$, which means deferred update is applied on the ϱ_i. To symbolize the naive version of deferred update we say $\curvearrowright^N$

Finally to make the problem abstract and easy to reason let me from here on talk only from interval perspective.

Scenario 1: *lender interval is not the only lender.* In *Algorithm 2* the deferred update happens with an assumption that lender,l of the relation window ϱ_i. This is generally not the case; There could be an interval, $I_{\varrho_i,btw}$ in between ϱ_i that could partially satisfy the capacity constraint of some interval, $I_{\varrho_i,aft}$ after the $I_{\varrho_i,btw}$ within ϱ_i.

Example: existence of multiple lenders within ϱ_i

To make the explanation more justifiable and apparent, I will first derive the offline version of spare capacity calculation in 2 steps.

TABLE I: Offline calculation of spare capacity

	Interval-id				
step	1	2	3	4	function applied
1	4	1	1	-3	$T_{\omega_i} \leftarrow \lvert\omega_i\rvert - \sum_{i=0}^{n} C_{J_{i,k}}$
2	3	-1	-2	-3	$\omega_i \leftarrow T_{\omega_i} + \min(0, \omega_{i+1})$

The above offline calculation of ω_i gives an simple insight on how $I_{\varrho_i,btw}$, .i.e. I_2 *and* I_3, partially satisfies capacity constraint of $I_{\varrho_i,aft}$, i.e. I_4. Lets assume deferred update offline phase is applied on this set of intervals and we label this relation window as ϱ_1 In this example for simplicity lets assume I_{curr} starts at I_1 and I_4 is the last interval. Now lets look into online phase of spare capacity maintenance with Algorithm2 and Algorithm 3 applied.

TABLE II: Online phase using deferred update

	Interval-id				
slot	1	2	3	4	function-applied
0	3	-1	-2	-3	$\twoheadrightarrow_{curr} I_1$
1	3	-1	-2	$-2^{[1]}$	$I_4 \twoheadrightarrow^{①} I_{curr}$
2	3	-1	-2	$-1^{[2]}$	$I_4 \twoheadrightarrow^{①} I_{curr}$
3	3	-1	-2	$0^{[3]}$	$I_4 \twoheadrightarrow^{①} I_{curr}$
4	2	-1	-2	$0^{[3]}$	$\emptyset \twoheadrightarrow^{①} I_{curr}$
5	2	2	1	0	$\twoheadrightarrow_{curr} I_2$ and $\curvearrowright^{N} \varrho_1$

but, this calculation is wrong in ω_1 and ω_2, lets run the same scenario with traditional algorithm, i.e. Algorithm 1 to understand what went wrong in the TABLE 2.

TABLE III: Online phase using Traditional algorithm

	Interval-id				
slot	1	2	3	4	function-applied
0	3	-1	-2	-3	$\twoheadrightarrow_{curr} I_1$
1	3	0	-1	-2	$I_4 \twoheadrightarrow I_{curr}$
2	2	1	0	-1	$I_4 \twoheadrightarrow I_{curr}$
3	1	1	1	0	$I_4 \twoheadrightarrow I_{curr}$
4	0	1	1	0	$\emptyset \twoheadrightarrow I_{curr}$
5	0	1	1	0	$\twoheadrightarrow_{curr} I_2$

On running naive-deferred-update we find ω_2 has 1 spare capacity extra and ω_1 has 2 spare capacity extra than the traditional algorithm. This additional spare capacity is an **error** caused due to naive-deferred-update.

scenario 2: *In online phase between ϱ_i some intervals becomes positive.*During online phase of deferred update irrespective of multiple lenders there could be an interval $I_{\varrho_i,btw}$ that becomes positive during O1-update-SC and before deferred-update is applied. To explain this scenario lets make an example.

Example: $I_{\varrho_i,btw}$ becomes positive during O1-update-SC and before deferred-update

To make this scenario simple and easy to understand lets make an assumption that lender l is the only interval that has lent to all the intervals in relation window ϱ_1 and ϱ_1 is the only set of intervals generated.

TABLE IV: Offline calculation of spare capacity

	Interval-id				
step	1	2	3	4	function applied
1	7	-1	-1	-3	$T_{\omega_i} \leftarrow \lvert\omega_i\rvert - \sum_{i=0}^{n} C_{J_{i,k}}$
2	2	-5	-4	-3	$\omega_i \leftarrow T_{\omega_i} + \min(0, \omega_{i+1})$

TABLE V: Online phase using deferred update

	Interval-id				
slot	1	2	3	4	function-applied
0	2	-5	-4	-3	$\twoheadrightarrow_{curr} I_1$
1	2	$-4^{[1]}$	-4	-3	$I_2 \twoheadrightarrow^{①} I_{curr}$
2	2	$-3^{[2]}$	-4	-3	$I_2 \twoheadrightarrow^{①} I_{curr}$
3	2	$-2^{[3]}$	-4	-3	$I_2 \twoheadrightarrow^{①} I_{curr}$
4	2	$-1^{[4]}$	-4	-3	$I_2 \twoheadrightarrow^{①} I_{curr}$
5	2	$0^{[5]}$	-4	-3	$I_2 \twoheadrightarrow^{①} I_{curr}$
6	1	$1^{[5]}$	-4	-3	$I_2 \twoheadrightarrow^{①} I_{curr}$
7	1	$1^{[5]}$	$-3^{[1]}$	-3	$I_3 \twoheadrightarrow^{①} I_{curr}$
8	1	2	-3	-3	$\twoheadrightarrow_{curr} I_2$ and $\curvearrowright^{N} \varrho_1$

On execution of I_3 in I_{curr}; I_{curr} should be negated by one because I_2 has already become positive so *neutralisation effect* will stop at I_2. This problem is due to the fact that I_3 does not know that I_1 has already become positive. this is an **error** that needs to be addressed in naive-deferred-update.

The above mentioned scenarios can occur in any combinations.

Understanding the Problem with naive deferred update: In above mentioned scenarios there is an error in deferred update due to the assumption that lender is the only lender and no interval within the ϱ_i will become positive when executing in I_{curr}. These problems arises because the function $\twoheadrightarrow^{①}$ is not aware of its past intervals which was not the case with traditional approach. Making the intervals become aware of each other is only possible by traversing backwards and updating the past intervals on each schedule which would de-vast the purpose of this approach. To address the problem we need to rectify the error during deferred update $\curvearrowright^{N} \varrho_i$.we call the error correction as *ripple effect correction(REC)*, ε.

The solution: To correct the error we now define ε in 2 steps each correcting error due to different scenarios and we conglomerate them into one single solution and apply them on our naive-deferred-update to make it into a complete working scenario.

Addressing Scenario 1. The scenario 1 leads to an error due to some intervals which were partial lenders becomes positive during deferred-update and still U_i of past intervals traverses forward. To address this scenario we define ε as

$$\varepsilon_c \leftarrow \max(0, ([\omega_c - \sum_{j=b_i-1}^{c+1} \varepsilon_j] + \sum_{k=b_i}^{c+1} U_k)) \tag{3}$$

so the spare capacity of the interval that needs to get updated becomes

$$\omega_c \leftarrow ([\omega_c - \sum_{j=b_i-1}^{c+1} \varepsilon_j] + \sum_{k=b_i}^{c+1} U_k) \tag{4}$$

- $c \leftarrow$ *is the current interval whose* ω *getting updated.*
- $j \leftarrow$ *ranges from interval before lent-till,* $b_i - 1$ *till the interval just after c within* ϱ_i.
- $k \leftarrow$ *ranges from lent-till,*b_i *till the interval just after c.*

when we look at the above formulation closer ω and ε are the variant of same formulae applied iteratively during $\curvearrowright \varrho_i$. The REC in this scenario is just the summation of all ω_i that is ≥ 0.

Addressing scenario 2. To make the solution easier to understand lets understand the cause for error. In offline phase the negative intervals in between ϱ_i also been added with a value in negative to its spare capacity ω_{btw}. The error is due to assumption that I_{ϱ_i} will not become positive before deferred update which is addressed in equations 3 and 4, but only moving this assumption is wrong. An interval in between $I_{\varrho_i,btw}$ can become positive satisfying the borrowed values of $I_{\varrho_i,aft}$, so any intervals $I_{\varrho_i,aft}$ executes in I_{curr} its U should not propagate beyond $I_{\varrho i,btw}$. This problem can be addressed by making every interval beware of other interval that becomes positive during execution, but on doing this $O(1)$ notion will be abolished, rather the whole U accumulated after $I_{\varrho_i,btw}$ can be made into an error to $I_{\varrho_i,bfr}$.i.e.

$$\varepsilon_c \leftarrow [\omega_{c,before} \geq 0]? \sum_{k=b_i}^{c+1} U_k : \sum_{j=b_i-1}^{c+1} \varepsilon_j \tag{5}$$

but ω_c update becomes same as equation 4.

- $c \leftarrow$ *is the current interval whose* ω *getting updated.*
- $j \leftarrow$ *ranges from interval before lent-till,* $b_i - 1$ *till the interval just after c within* ϱ_i.
- $k \leftarrow$ *ranges from lent-till,*b_i *till the interval just after c.*
- $\omega_{c,before}$ *is the spare capacity of the interval before deferred update on the interval c.*

Addressing Current interval I_{curr}. The current interval I_{curr} need not be added with update-val U rather just needs to be corrected with REC, ε collected from previous intervals .i.e.

$$I_{curr}.\omega \leftarrow I_{curr}.\omega - \sum_{j=b_i-1}^{curr+1} \varepsilon_j \tag{6}$$

- $j \leftarrow$ *ranges from interval before lent-till,* $b_i - 1$ *till the interval just after* I_{curr} *within* ϱ_i.

Conglomerated solution. On conglomerating above 3 solutions into one single algorithm leads to complete deferred update algorithm which is as follows.

Algorithm 3: deferred-update

Function deferred-update(I, I_{curr});
Input : I: Interval list, I_{curr}: Current interval
Result: spare capacity of the interval within ϱ_i is updated with right data

$lentTill_{temp} \leftarrow I_{curr}.b$;
if $!lentTill_{temp}$ **then**
  return;
end
$updateVal \leftarrow 0$;
$\varepsilon \leftarrow 0$;
while $lentTill_{temp} \; != I_{curr}$ **do**
  $\omega_{before} \leftarrow lentTill_{temp}.\omega$ ▷ Make a copy of ω before updating;
  $lentTill_{temp}.\omega \leftarrow lentTill_{temp}.\omega + updateVal$ ▷ Update the deferred spare capacity;
  if $\omega_{before} < 0$ *and* $lentTill_{temp} \; != I_{curr}.b$ **then**
    $\varepsilon \leftarrow \varepsilon + \max(0, lentTill_{temp}.\omega)$
  end
  else
    $\omega \leftarrow updateVal$
  end
  $updateVal \leftarrow updateVal + lentTill.U$ ▷ Accumulate previous intervals update-val within ϱ_i ;
  $lentTill.U \leftarrow 0$;
  $lentTill \leftarrow lentTill.prev$;
end

B. Removing Slots

To remove the notion of slots we need to make the following alterations along with the existing one.

- we just need to make the tasks continuous, i.e. we need to remove the additional computation mentioned in *equation 2* and compute the start and end of intervals with the notion of continuous time.
- Decision timer that gets triggered for every slot needs to be triggered here on at the end of the interval.i.e.

$$\tau_{expiry} \leftarrow I_{curr}.end \tag{7}$$

- we need to alter the slot based O1-Update-SC into capacity based O1-Update-SC which is as mentioned in Algorithm 5.

Algorithm 4: Capacity Based O1-Update-SC

Function o1-update-sc($J_{i,k,curr}, I$);
Input : $J_{i,k,curr}$: Previous selected job, I: Interval table
Output: None
Result: Checks and Updates only spare capacity of job and current interval

if $J_{i,k,curr}.interval == I_{curr}$ **then**
  return;
end
▷ check for 1^{st} time and initialize $sched-time$;
if $first-time$ **then**
  $sched-time \leftarrow 0$;
  $first-time \leftarrow 0$;
end

$run-time \leftarrow [curr-time]-[sched-time]$ ▷ compute the difference between previous and current schedule time;
$sched-time \leftarrow curr-time$;
$tsk-\omega \leftarrow J_{i,k,curr}.interval.\omega$ ▷ take a local copy of task's spare capacity before updating;
if $J_{i,k,curr}.interval.\omega < 0$ **then**
  ▷ here just update update-val with required value;
  $J_{i,k,curr}.interval.\omega \; += run-time$;
  if $J_{i,k,curr}.interval.\omega > 0$ **then**
    $J_{i,k,curr}.interval.u \leftarrow J_{i,k,curr}.interval.u + [[run-time]-[J_{i,k,curr}.interval.\omega]]$;
  end
  else
    $J_{i,k,curr}.interval.u \leftarrow J_{i,k,curr}.interval.u + run-time$;
  end
end
else
  $J_{i,k,curr}.interval.\omega \; += run-time$;
end

▷ check job's lender and current interval's lender is same and job's interval before updating is negative;
if $J_{i,k,curr}.interval.l == I_{curr}.l \quad and \quad tsk-\omega < 0$ **then**
  return;
end
$I_{curr}.\omega \; -= run-time$;

Notion in words, since the notion of slot is removed and the task can run in I_{curr} continuously even beyond the ω that was borrowed, so we need to update $u_{J_{i,k,curr}}$only the required, so remaining should be avoided hence derives the following equation.

$$J_{i,k,curr}.I.\omega \leftarrow J_{i,k,curr}.I.\omega + R \tag{8}$$

$$J_{i,k,curr}.I.u \leftarrow [J_{i,k,curr}.I.\omega > 0]?(J_{i,k,curr}.I.u + [[R] - [J_{i,k,curr}.I.\omega]]) : (J_{i,k,curr}.I.u + R) \tag{9}$$

- $R \leftarrow$ runtime between previous schedule and current schedule

With the mentioned algorithm **natural process of decision** happens except at the end of interval trigger. The decision happens during the following scenarios.

- when a job arrives
- when a job exits
- End of the interval,i.e. when $Timer$, τ expires

We would symbolize the capacity based O1-Update-SC as $\twoheadrightarrow^{C}$ with the same notion as $\twoheadrightarrow^{①}$, but rather than Algorithm 2, Algorithm 4 will be applied.

C. Guarantee Algorithm

To address the problems mentioned in *section 2 problem 3* we derive a new guarantee algorithm that just traverses the intervals and just updates only spare capacity rather than recalculation of spare capacity on applying *formulae 1.* Moreover the traversal stops at the point where the C_i is completely satisfied.

Algorithm 5: New Guarantee Algorithm

```
1  Function Guarantee-algorithm(I, I_acc, Γ);
   Input  : I: list of intervals;
            I_acc: reference of the interval at which Γ deadline falls;
            Γ : The aperiodic job that got accepted
   Output: None
   Result: Γ is guaranteed
2  if I_acc.end > Γ.C then
3  |  I_New = split-intr(I_acc, Γ.C);
4  end
5
6  if I_New then
7  |  I_iter ← I_New;
8  end
9  else
10 |  I_iter ← I_acc
11 end
12 Δ ← Γ.C;
13 while Δ do
14 |  if I_iter.ω < 0 then
15 |  |  I_iter.ω ← I_iter.ω − Δ
16 |  end
17 |  else if I_iter.ω > 0 then
18 |  |  if I_iter.ω ≥ Δ then
19 |  |  |  I_iter.ω ← I_iter.ω − Δ;
20 |  |  |  Δ ← 0;
21 |  |  end
22 |  |  else
23 |  |  |  I_iter.ω ← I_iter.ω − Δ;
24 |  |  |  Δ ← Δ − I_iter.ω;
25 |  |  end
26 |  end
27 |  I_iter ← I_iter.prev;
28 end

29 Function split-intr(I_right,sp);
   Input  : I_right: reference of the interval at which Gamma falls;
            sp: split point at which separation should happen
   Output: I_left: New interval that was created and added to list

30            I_left.start ← I_right.start;
31            I_left.end ← sp;
32            I_left.sc ← I_left.end − I_left.start;
33            I_right.start ← sp;
34            I_right.sc ← I_right.sc − I_left.sc;
35            I_left.sc ← I_left.sc + min(0, I_right.sc);
36            insert-before(I_right, I_left)        ▷ list function that just inserts the node before first parameter;
37            return I_left
```

In brief, The algorithm uses a notion called delta, Δ which will initially hold the $\Gamma.c$ and updates the intervals till the Δ becomes 0. Negation of Δ happens in the following steps as we traverse backwards from either I_{new}(in case

a new interval is created) or I_{acc},i.e the interval where deadline falls. The idea behind the computation is to use the existing spare capacity calculation of offline phase and just update new spare capacity with the same ideology but from different perspective.

- when the interval ω is positive and less than required Δ then negate the Δ with ω and make ω the negative version of the remaining Δ.
- when the interval's ω is positive and greater or equal to required Δ then just negate ω with Δ and make Δ to 0.
- Finally when interval's ω is negative then simply negate Δ to ω,i.e. $\omega \leftarrow \omega - \Delta$

In case of new interval creation we start from I_{new} because the ω of the I_{acc} is already satisfied during split-interval function.

D. Putting it all together

This section just tries to show how the capacity algorithm will incorporate into the online phase of traditional Algorithm. When we notice the algorithm the notion of slot, s and slot count s_{cnt} is completely removed.

Algorithm 6: Capacity Shifting

Function schedule($J_{i,k,Curr}$);
Input : $J_{i,k,Curr}$: job previously selected
Output: $J_{i,k,nxt}$: next selected job
O1-Update-SC($J_{i,k,Curr}, I$) ▷ update the intervals with the run time;
if ι_A *!= None* **then**
  deferred-update(I, I_{curr}) ▷ Deferred update happens;
  Test-aperiodic(ι_A) ▷ Test and try Guarantee Aperiodic's if any;
end
update-time() ▷ Here deferred update happens if needed;
$J_{i,k,nxt}$ ←selection-fn() ▷ Get next eligible job;
return $J_{i,k,nxt}$;

Function o1-update-sc($J_{i,k,curr}, I$);
Input : $J_{i,k,curr}$: Previous selected job, I: Interval table
Output: None
Result: Checks and Updates only spare capacity of job and current interval

if $J_{i,k,curr}.interval == I_{curr}$ **then**
  return;
end
▷ check for 1^{st} time and initialize $sched - time$;
if $first - time$ **then**
  $sched - time \leftarrow 0$;
  $first - time \leftarrow 0$;
end

$run - time \leftarrow [curr - time] - [sched - time]$ ▷ compute the difference between previous and current schedule time;
$sched - time \leftarrow curr - time$;
$tsk - \omega \leftarrow J_{i,k,curr}.interval.\omega$ ▷ take a local copy of task's spare capacity before updating;
if $J_{i,k,curr}.interval.\omega < 0$ **then**
  ▷ here just update update-val with required value;
  $J_{i,k,curr}.interval.\omega \; += run - time$;
  if $J_{i,k,curr}.interval.\omega > 0$ **then**
    $J_{i,k,curr}.interval.u \leftarrow J_{i,k,curr}.interval.u + [[run - time] - [J_{i,k,curr}.interval.\omega]]$;
  end
  else
    $J_{i,k,curr}.interval.u \leftarrow J_{i,k,curr}.interval.u + run - time$;
  end
end
else
  $J_{i,k,curr}.interval.\omega \; += run - time$;
end

▷ check job's lender and current interval's lender is same and job's interval before updating is negative;
if $J_{i,k,curr}.interval.l == I_{curr}.l \quad and \quad tsk - \omega < 0$ **then**
  return;
end
$I_{curr}.\omega \quad -= run - time$;
Function update-time();
Input : $None$
Output: $None$
Result: checks and moves I_{curr} to $I_{curr}.nxt$ along with deferred-update
if $get\text{-}curr\text{-}time() \geq I_{curr}.end$ **then**
  deferred-update(I,I_{curr});
  $I_{curr} \leftarrow I_{curr}.nxt$;
end

Function Test-aperiodic(ι_A, I);
Input : ι_A: Unconcluded firm aperiodic list;
I: list of intervals
Output: $None$
Result: Job in ι_A is moved to either ι_G or ι_S
while ι_A != $None$ **do**
  $\Gamma \leftarrow A.dequeue()$;
  if $I_{acc} \leftarrow Acceptance\text{-}test(\Gamma, I)$!= $None$ **then**
    Guarantee-job(I, I_{acc}, Γ);
    $\iota_G.queue(\Gamma)$
  end
  else
    $\iota_S.queue(\Gamma)$
  end
end

Function deferred-update(I, I_{curr});
Input : I: Interval list, I_{curr}: Current interval
Result: spare capacity of the interval within ϱ_i is updated with right data

$lentTill_{temp} \leftarrow I_{curr}.b$;
if $!lentTill_{temp}$ **then**
  return;
end
$updateVal \leftarrow 0$;
$\varepsilon \leftarrow 0$;
while $lentTill_{temp} \quad != I_{curr}$ **do**
  $\omega_{before} \leftarrow lentTill_{temp}.\omega$ ▷ Make a copy of ω before updating;
  $lentTill_{temp}.\omega \leftarrow lentTill_{temp}.\omega + updateVal$ ▷ Update the deferred spare capacity;
  if $\omega_{before} < 0$ and $lentTill_{temp} \quad != I_{curr}.b$ **then**
    $\varepsilon \leftarrow \varepsilon + \max(0, lentTill_{temp}.\omega)$
  end
  else
    $\omega \leftarrow updateVal$
  end
  $updateVal \leftarrow updateVal + lentTill.U$ ▷ Accumulate previous intervals update-val within ϱ_i ;
  $lentTill.U \leftarrow 0$;
  $lentTill \leftarrow lentTill.prev$;
end

```
1  Function Acceptance-test(Γ, I);
   Input  : Γ: Unconcluded firm aperiodic job;
            I: list of intervals
   Output: I_acc: interval in which task was accepted or None
2  I_tmp ← I_curr;
3  ω_tmp ← 0;
4  while I_tmp ≤ Γ.d do
5  |  if I_tmp.ω > 0 then
6  |  |  ω_tmp ← I_tmp.ω + ω_tmp;
7  |  end
8  |  I_tmp ← I_tmp.nxt;
9  end
10 if ω_tmp ≥ Γ.C then
11 |  return I_tmp.prev;
12 end
13 else
14 |  return None;
15 end

16 /* New Version of guarantee algorithm;
17 Notice has only one loop and stops at a point Δ satisfies;
18 */;
19 Function Guarantee-job(I, I_acc, Γ);
   Input  :  I: list of intervals;
             I_acc: reference of the interval at which Γ deadline falls;
             Γ : The aperiodic job that got accepted
   Output: None
   Result: Γ is guaranteed
20 if I_acc.end > Γ.C then
21 |  I_New = split-intr(I_acc, Γ.C);
22 end
23
24 if I_New then
25 |  I_iter ← I_New;
26 end
27 else
28 |  I_iter ← I_acc
29 end
30 Δ ← Γ.C;
31 while Δ do
32 |  if I_iter.ω < 0 then
33 |  |  I_iter.ω ← I_iter.ω − Δ
34 |  end
35 |  else if I_iter.ω > 0 then
36 |  |  if I_iter.ω ≥ Δ then
37 |  |  |  I_iter.ω ← I_iter.ω − Δ;
38 |  |  |  Δ ← 0;
39 |  |  end
40 |  |  else
41 |  |  |  I_iter.ω ← I_iter.ω − Δ;
42 |  |  |  Δ ← Δ − I_iter.ω;
43 |  |  end
44 |  end
45 |  I_iter ← I_iter.prev;
46 end
```

1 **Function** split-intr(I_{right},sp);
Input : I_{right}: reference of the interval at which *Gamma* falls;
sp: split point at which separation should happen
Output: I_{left}: New interval that was created and added to list

2 $I_{left}.start \leftarrow I_{right}.start$;
3 $I_{left}.end \leftarrow sp$;
4 $I_{left}.sc \leftarrow I_{left}.end - I_{left}.start$;
5 $I_{right}.start \leftarrow sp$;
6 $I_{right}.sc \leftarrow I_{right}.sc - I_{left}.sc$;
7 $I_{left}.sc \leftarrow I_{left}.sc + \min(0, I_{right}.sc)$;
8 insert-before(I_{right}, I_{left}) ▷ list function that just inserts the node before first parameter;
9 return I_{left}

Algorithm 6 is just abstract conglomeration of all the explanations in this paper making the reader easy to understand the whole flow of online phase. When we notice in scenarios like aperiodic arrival deferred-update might happen twice where the 2nd iteration would be a simply dumb loop, which could be avoided by simple conditional flag checks, but I haven't made it part of my algorithm just to keep the gist of the algorithm simple and understandable.

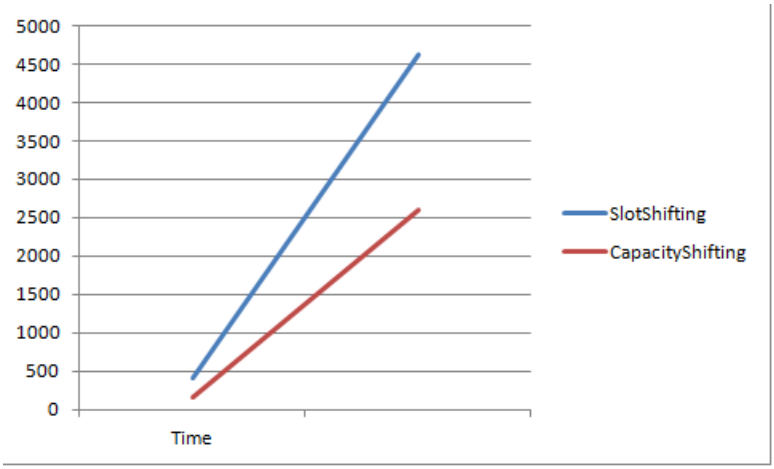
When the whole algorithm is noticed the complexity of scheduling selection within interval due to notion of updating spare capacity is made $O(1)$, but the interval movement still holds a complexity of $O(n)$, where n is the number of intervals in the ϱ_i, but when we notice the traditional algorithm had the same complexity per slot, which we reduced to per interval.

Guarantee-job. when the traditional algorithm is noticed complexity of recalculation of spare capacity per interval during guarantee algorithm is $O(n)$, where n is the number of job's in that interval. We in the proposed algorithm reduced this complexity to $O(1)$ by just either adding or negating the computation time of the Γ to the ω of the interval.

V. Experimental Results

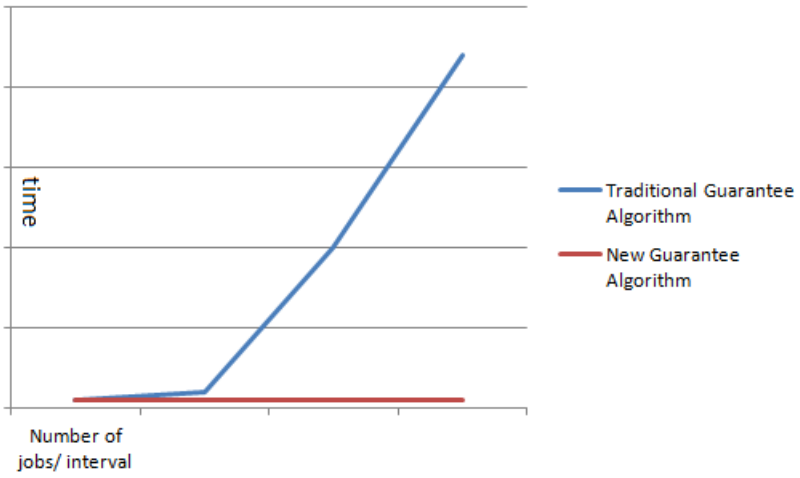
Online and offline phase of both Slot shifting and Capacity shifting was implemented in SimSo simulator framework. To make the whole implementation work very abstract and adaptable to changes with respective slot shifting; A generic abstract framework was created within SimSo to adapt any future changes with respect to Slot shifting. The framework also assumes that all jobs runs exactly for its computation time. The tasksets are randomly generated using **ripoll and uunifast generators**. Maximum of upto 5000 jobs are used to test the correctness and benchmark between traditional and new algorithm's computational need.

Fig. 1: shows the number of scheduling decisions taken with respective to Capacity shifting against slot shifting



The Y axis shows the number of scheduling decisions taken on an average 2604 jobs with 4650 slots decisions in slot shifting against capacity shifting which takes on an average of 45 percent less scheduling decision with respect to slot shifting and in many best cases goes to 60 percentage.

Fig. 2: Shows that traditional guarantee algorithm that time increases as number of jobs per interval increases



The Y axis shows the computational time per interval and we can notice a proportional increase in computation time as number of jobs increases in traditional algorithm against proposed guarantee algorithm which remains constant.

VI. Conclusion and future work

In this paper we have derived a new form of algorithm called Capacity shifting which is inspired and refereed from Slot shifting. This is a novel approach that removes the notion of slot reducing the schedule computation by 60 percent. Moreover we also derived a new version of Guarantee algorithm that showed an constant or $O(1)$ performance per interval irrespective of the number of jobs per interval.

Moreover following are some more areas of observation and to cover with respect to improvising Slot shifting.

- **compute when required.** I tried to defer the computation of spare capacity to the end and beginning of intervals, but this could further be extended to scenario where the spare capacity computation would be computed on need,i.e. when Firm aperiodic's arrive.
- The capacity shifting could be extended to multicore computations.
- The deferring spare capacity update happens irrespective of update is required or not , this could be avoided by small conditional checks.
- Problem and issues
  - A small drift in clock in the system will cause the whole system failure.
  - The Slotshifting was implemented by me in a realtime platform like $LITMUS^{RT}$ and also in a simulation environment like SimSo, The major problem was injection of precomputed tables of jobs and intervals, though tried to keep representation of data in compressed format , The problem of injection of such a table into system is a huge effort with respective to both implementation and online preparation phase of the system. just to glimpse In some cases of non-harmonic task set just a count 10 tasks produces interval tables in count of 1000's for a hyper period.

Acknowledgment

I would like to personally thank John Gamboa for being on my side to derive the new Guarantee Algorithm. I would like to thank Mitra Nasri for being patient whenever i came up with some whacky ideas of implementation. Finally My professor Gerhard Fohler for deriving a beautiful algorithm called Slot shifting and giving me this opportunity to implement the new version of it.

References

[1] [FH94] Gerhard fohler, Flexibility in Statically Scheduled Hard Real-Time Systems. The dissertation on Slot shifting algorithm.

[2] Implementation of Capacity shifting and Slot shifting in Simso.
`https://github.com/gokulvasan/CapacityShifting`

[3] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. Real-Time System., Volume 29, Issue 1:5-26, January 2005.

[4] Gowri Sankar Ramachandran, Integrating enhanced slot-shifting in μC/OS-II.School of Innovation, Design and Engineering, Malardalen University, Sweden.

[5] Giorgio Buttazzo's. HARD REAL-TIME COMPUTING SYSTEMS Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park Norwell, MA 02061, USA, 1997.

[6] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. Real-Time Systems: The International Journal of Time-Critical Computing, 11:19–39, 1996.

[7] .E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. Real-Time Systems, 30(1-2):129–154, 2005.

[8] G. C. Buttazzo and E. Bini. Optimal dimensioning of a constant bandwidth server. In Proc. of the IEEE 27th Real-Time Systems Symposium (RTSS 2006), Rio de Janeiro, Brasil, December 6-8, 2006.

[9] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In Proceedings of the 16th Real-Time Systems Symposium, Pisa, Italy, December 1995.

[10] G.Fohler, Analyzing a Pre Run-Time Scheduling Algorithm and Precedence Graphs, Research Report Institut fur Technische Informatik Technische Universitat Wien, Vienna, Austria ,September 1992

[11] C. Lin and S. A. Brandt. Improving soft real-time performance through better slack management. In Proc. of the IEEE Real-Time Systems Symposium (RTSS 2005), Miami, Florida, USA, December 5, 2005.

[12] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In Proceedings of the IEEE Real-Time Systems Symposium, December 1993.

[13] Maxime Ch´eramy, Pierre-Emmanuel Hladik, Anne-Marie D´eplanche, SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms.5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), Jul 2014, Madrid, Spain. 6 p., 2014

[14] Simso simulator framework official website.
`http://projects.laas.fr/simso`

[15] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, Pennsylvania, USA, May 1993

[16] M. Spuri and G. C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In Proceedings of the IEEE Real-Time Systems Symposium, December 1994.

[17] M. Spuri and G. C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. Journal of Real-Time Systems, 10(2), 1996.

[18] Fabian Scheler and Wolfgang Schroeder-Preikschat. Time-triggered vs. event-triggered: A matter of conguration? Model-Based Testing, ITGA FA 6.2 Workshop on and GI/ITG Workshop on Non-Functional Properties of Embedded Systems, 2006 13th GI/ITG Conference -Measuring, Modelling and Evaluation of Computer and Communication (MMB Workshop), pages 1-6, March 2006.

[19] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Pre-emptively scheduling hard-real-time sporadic tasks on one processor. In Proc. Real-Time Systems Symposium (RTSS), pages 182-190, 1990.

[20] Damir Isovic. Flexible Scheduling for Media Processing in Resource Constrained Real-Time Systems. PhD thesis, Malardalen University, Sweden, November 2004.

[21] Damir Isovic and Gerhard Fohler. Handling mixed task sets in combined offine and online scheduled real-time systems. Real-Time Systems Journal, Volume 43, Issue 3:296-325,December 2009.

[22] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, pages 87-101, London, UK, 1991. Springer-Verlag.

[23] Martijn M. H. P. van den Heuvel, Mike Holenderski, Reinder J. Bril, and Johan J.Lukkien. Constant-bandwidth supply for priority processing. In IEEE Transactions on Consumer Electronics (TCE), volume 57, pages 873-881, May 2011.