An Efficient Implementation of Slot Shifting Algorithm based on deferred Update

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

Dr.Mitra Nasri
Max Plank Institute for Software Systems
Kaiserslautern, Germany
Email: mitra@mpi-sws.org

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

Abstract

Slot-shifting algorithm provides a mechanism to accommodate event triggered tasks along with periodic tasks. Slot-shifting uses the residual bandwidth of the periodic tasks to accommodate aperiodic tasks, called *spare capacity*, to improve the chances of admitting aperiodic tasks. In contrast to bandwidth servers, slot-shifting preserves the unoccupied capacity of the system by shifting the spare capacity to future.

Slot-shifting uses a discrete time model, termed as slots. Slot-shifting assumes a global time, whose progression is triggered by equidistant events, detected by an independent external observer. At the beginning of each slot the scheduler is invoked to determine which job of which task to schedule in that slot. The scheduler will also update the spare capacity of the system. This activation of scheduler for each slot is computationally expensive and has a considerable overhead. In this work we will provide an approach to remove the slots and replace it with a larger quantity that is naturally related to the release time and deadlines of the jobs. Moreover, we provide a method for delayed updates of the space capacities in order to reduce the runtime computational complexity, but preserving the notion incepted by slot shifting. We will show that this new approach reduces the scheduling overhead of slot shifting by around *60%* in best cases and *45%* on average cases.

Finally, The work will provide a new approach to guarantee the aperiodic task. This new solution will have time complexity of *O(1)* per *interval* compared to existing (slot-shifting) algorithm having complexity of *O(n)* per interval, where n is the complexity involved to iterate over n tasks that belongs to an interval.

I. Introduction

Real-time systems must fulfill twofold constraints, in order to be considered working functionally correct: First, they must process information and consecutively produce a correct output behaviour. Second, their interaction with the environment must happen within stringent timing constraints dictated by the corresponding environment. In contrast to many other systems, real-time systems are thus not primarily optimized for speed (in the sense of maximized for throughput), but to provide determinism and worst case guarantees.

To ensure meeting their strict timing constraints, real-time systems utilize real-time scheduling algorithms. A scheduling algorithm determines the execution order of the jobs of the workload on the processors of the system. There exist many different classification schemes for scheduling algorithms, e.g., depending on the method to prioritize different jobs, or depending on the moment in time when scheduling decisions are made. An important classification of scheduling algorithms is time-triggered and event-triggered algorithms. Event-triggered algorithms are based on a set of rules that are used at runtime of the system to make the scheduling decisions. In contrast to this, time-triggered algorithms determine the execution order of the jobs statically, i.e., prior to the runtime of the system. Static scheduling has been shown to be appropriate for a variety of hard real time systems mainly due to the verifiable timing behaviour of the system and the complex task models supported. Another classification of scheduling algorithms is established on the criterion by which priorities are assigned to the tasks. If each task is assigned a fixed priority that does not change at runtime from job to job of the same task, then this is called fixed (task) priority scheduling . If the priority of jobs of the same task may change over time or from job to job, this is called dynamic task priority scheduling.

Most of real time applications have both periodic tasks and aperiodic tasks. Typically, periodic tasks are time driven and execute the demanding activities with hard timing constraints aimed at guaranteeing regular activation rates. Aperiodic tasks are usually event driven and may have hard, soft, or non real time requirements depending on the specific application. When dealing with hybrid task sets, the main objective of the system is to assure the schedulability of all guaranteed tasks in worst case conditions and provide good average response times for soft and non-realtime activities. Offline guarantee of event driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment, i.e., by assuming the maximum arrival rate for each critical event. If the maximum arrival rate of some event cannot be bounded a priori, the associated aperiodic task cannot be guaranteed statically, although an online guarantee of individual aperiodic requests can still be done. Aperiodic tasks requiring

online guarantee on individual instances are called *firm*. Whenever a firm aperiodic request enters the system, an acceptance test can be executed by the kernel to verify whether the request can be served within its deadline. If such a guarantee cannot be done, the request is rejected. On rejection of firm aperiodic tasks the systems predictability or deterministic behaviour will not be effected and will not cause safety hazard for the system.

A. Related Work

Server algorithms for fixed priority scheduling [2, 3, 4], as well as for dynamic priority scheduling [5, 6], aim at reserving a fraction of the processor bandwidth to the aperiodic jobs. The server algorithms introduce an additional periodic task, the server task, into the schedule. The purpose of the periodic server task is to service aperiodic requests as soon as possible. Like any periodic task, a server is characterized by a period T_s and a computation time C_s called **server capacity**.

Polling Server, (PS) [2]. At periods T_s server becomes active and serves aperiodic requests with its capacity C_s. If no aperiodic requests are pending, PS suspends itself until the beginning of its next period, and the time originally allocated for aperiodic service is not preserved for aperiodic execution but is used by periodic tasks. If no aperiodic task arrives the capacity is wasted.Note that if an aperiodic request arrives just after the server has suspended, it must wait until the beginning of the next polling period, when the server capacity is replenished at its full value. The server is based on Rate Monotonic Scheduling

Deferrable Server, (DS) [2,3].DS algorithm creates a periodic task (usually having a high priority) for servicing aperiodic requests. DS preserves its capacity if no requests are pending upon the invocation of the server. The capacity is maintained until the end of the period, so that aperiodic requests can be serviced at the same server's priority at any-time, as long as the capacity has not been exhausted. At the beginning of any server period, the capacity is replenished at its full value.

Priority Exchange [2,3].The algorithm uses a periodic server (usually at a high priority) for servicing aperiodic requests.The server preserves its high-priority capacity by exchanging it for the execution time of a lower-priority periodic task. At the beginning of each server period, the capacity is replenished at its full value. If aperiodic requests are pending and the server is the ready task with the highest priority, then the requests are serviced using the available capacity; otherwise C_s is exchanged for the execution time of the active periodic task with the highest priority.When a priority exchange occurs between a periodic task and server, the periodic task executes at the priority level of the server while the server accumulates a capacity at the priority level of the periodic task. Thus, the periodic task advances its execution, and the server capacity is not lost but preserved at a lower priority.

Slack Stealing [4]. The Slack Stealing algorithm does not create a periodic server for aperiodic task service. Rather it creates a passive task, referred to as the Slack Stealer, which attempts to make time for servicing aperiodic tasks by "stealing" all the processing time it can from the periodic tasks without causing their deadlines to be missed. The main idea behind slack stealing is that, typically, there is no benefit in early completion of the periodic tasks. when an aperiodic request arrives, the Slack Stealer steals all the available slack from periodic tasks and uses it to execute aperiodic requests as soon as possible. If no aperiodic requests are pending, periodic tasks are normally scheduled by rate monotonic algorithm.

Constant Bandwidth Server [5]. When a new job enters the system, it is assigned a suitable scheduling deadline (to keep its demand within the reserved bandwidth) and it is inserted in the EDF ready queue. If the job tries to execute more than expected, its deadline is postponed (i.e., its priority is decreased) to reduce the interference on the other tasks. Note that by postponing the deadline, the task remains eligible for execution. In this way, the CBS behaves as a work conserving algorithm, exploiting the available slack in an efficient

Total Bandwidth Server [6]. The assignment must be done in such a way that the overall processor utilization of the aperiodic load never exceeds a specified maximum value. Each time an aperiodic request enters the system, the total bandwidth of the server is immediately assigned to it, whenever possible. Once the deadline is assigned, the request is inserted into the ready queue of the system and scheduled by EDF as any other periodic instance. when the kth aperiodic request arrives at time $t = r_k$, it receives a deadline. $d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s}$ where c_k is the execution time of the request and U_s is the server utilization factor(which is the bandwidth).

When noticed the bandwidth servers needs an explicit bandwidth allocation. The main drawback of this approach is that a substantial amount of the CPU utilization might be reserved for future aperiodic jobs that will not necessarily arrive. Another drawback is that using the server algorithm can result in a lower schedulability bound of the system. To the best of our knowledge, it has not been shown how to combine server algorithms with arbitrary time-triggered scheduling tables. Though bandwidth allocated, servers don't provide guarantee for the aperiodic task whose execution time is beyond C_s. Methods like slack stealing do try to admit the aperiodic tasks without explicit bandwidth, but

does not guarantee the admittance of firm aperiodic task. Moreover, the allocation of bandwidth utilisation in servers is very minimal, because of random arrival nature of events.

Slot shifting algorithm which combines the benefits of both time- and event-triggered scheduling for systems. Slot shifting resolves the complex constraints of a set of online tasks by constructing an static offine scheduling table. Similar to the aforementioned slack stealing algorithm, slot shifting expresses the leeway of tasks in this derived offine schedule by spare capacities. At runtime of the system, slot shifting performs acceptance tests for the individual jobs of aperiodic tasks and integrates them feasibly into the schedule. With the advantages also comes overhead on the system. This paper tries to address one among the prominent overheads, slots. We would address this problem by completely removing this notion, but still preserving the idea of slot shifting leading to a new notion of algorithm.

We will also present a new guarantee algorithm which has a much less latency, easy to implement and needs less data space for computation compared to existing slot shifting algorithm.

B. Organization of the report

The following work is organized as follows: The Section 2 provides the system-model and notations followed by description of slot shifting algorithm. The section 3 would provide an elaborate version of problem statement which would succor in establishing solution. Purposefully the problem statement is deferred after algorithmic explanation of slot shifting, because this might provide a profound realization of the presented complication. The section 4 would provide the proposed solution. Incremental solution is presented with problems behind each phase; Enabling the assayer understand the real intuitiveness behind the final solution. The section 5 would provide the experimental results followed by conclusion and future work.

II. System Model and Background

Task set. Consider a realtime system with n periodic tasks, $\tau = \{\tau_1, ..., \tau_n\}$. Each task τ_i has a worst case execution time (WCET) 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 the Earlies-Deadline-First (EDF) 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, generates 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 an optional hard deadline, D_k. The tasks that arrives with such hard deadline is called *firm aperiodic job*, Γ and tasks with no hard deadline is named *soft aperiodic job*, ζ, if the aperiodic job does not have a hard deadline, we set $D_k = \infty$. The aperiodic job are indexed such that $0 \leq \alpha_k \leq \alpha_{k+1}, K \geq 1$. We assume that the aperiodic job sequence is not known in advance. On arrival of Γ the job is temporarily placed on a list before acceptance test applied on it. The list that holds the set of firm aperiodic task waiting to be accepted is called *unresolved list*,ι_A. The list holding set of γ_k which could not be guaranteed and not a ζ is called *not guaranteed list*,ι_S, job from list,ι_S is named $J_{i,k,S}$. The list holding a set of jobs that is ready to run is called *ready list* ,ι_ξ. Irrespective of the list, Job that is selected and running in the system is called current job, $J_{i,k,curr}$.

Interval. A layer of certainty is added around the guaranteed jobs called interval. The definition of interval is as follows, Each interval has an id,i. The end of an interval, $e_i = d_{J_{i,k}}$. The end of an interval determines the owner or nativity of the jobs, 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

$$\omega_i = |\omega_i| - \sum_{i=0}^{n} C_{J_{i,k}} + \min(0, \omega_{i+1}). \tag{1}$$

Spare capacity and interval calculation notion could create a *negative spare capacity* interval. This could create a consecutive backward wave of negative spare capacity intervals, till an interval which could completely satisfy the capacity needed for such interval. This ripple effect due to backward propagation of negative spare capacity forms a relation among these intervals is denoted as ***relation window***, ϱ_i (Diagrammatic representation in Fig 1). The first

interval of this relation window is named ***lender***, l_i and the last in the relation window is named ***lent-till***, b_i. If interval is part of no relation window then $l_i = b_i = \infty$. There could be many such relation window in a system within its hyper period.

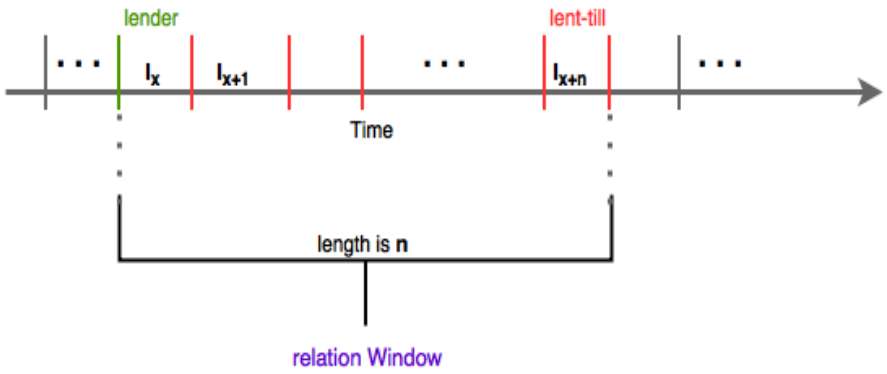

Fig. 1: Relation window

In general, each interval I_i is represented as a set $I_i = \{i, s, e, \omega, l, b\}$. Each element within the set is accessed using member access operator " . "(Dot operator [1]).

The interval progression is closely associated with time, the interval that is associated 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 of intervals is simply named I. Each job will hold the reference to the interval to which it belongs.

In general, each Job,$J_{i,k}$ is represented as a set $J_{i,k} = \{C, T, \phi, d, I\}$. Each element within the set is accessed using member access operator " . "(Dot operator [1]).

System widely uses the notion of list. The list in general can be described as an enumerated collection of objects. Like a set, it contains members. Unlike set the elements have position. The position of an element in a list is its *rank or index*. The number of elements is called the length of the sequence. Formally, a list can be defined as an abstract data type whose domain is either the set of the natural numbers (for infinite sequences) or the set of the first n natural numbers (for a sequence of finite length n). The logical behaviour is defined by a set of values and a set of operations. The order of arrangement is an abstract functionality, where *dequeue()* removes the element with rank one and *queue()* would add an element at appropriate rank. The list holds the current accessed element rank. The traversal of the list from the current rank i to $i-1$ is called *prev* operator. Similarly the traversal from current rank to i to $i-1$ is through *nxt* operator. In this work, behavioural access of the list is done through symbol " . "(Dot operator[1]).

During online phase, the **timer** λ is triggered based on the parameter λ_{expiry}, which in case of slot shifting is a fixed relative period called slots. A **slot** s is defined as an external observer (λ) counts the ticks of the globally synchronized clock with granularity of slot length,$|s|$, i.e., $\lambda_{expiry} \leftarrow |s|$ 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.

A. Background

Now we sketch the existing version of slot shifting algorithm. Describing the existing solution would enable us hand on hand comparison of the slot shifting algorithm with the proposed one, giving us intuitiveness of the problem and justifying the proposed solution.

The slot shifting algorithm works at two phases namely, offline phase and online phase. In **Offline Phase,** the periodic tasks are broken into its corresponding jobs and a layer of certainty is added around the jobs through interval, then the spare capacity for each interval is calculated with equation(1) (Example below gives diagrammatic interpretation). This yields to a table of jobs and corresponding intervals.

[1]same as ANSI C's dot operator

Example: Exemplification of spare capacity calculation[2]

The intervals are created by applying the formulation $e = d_{J_{i,k}}$ which calculates end of an interval, then the start of an interval is calculated using $s = max(e_{i-1}, \epsilon_i)$. The created intervals are assigned with unique identification by assigning the 1^{st} interval 1, 2^{nd} interval 2,...so on, i.e., single count integer increment value is assigned to each consecutive interval starting with value 1 for 1^{st} interval till the last interval. Along with the id assignment initial spare capacity is assigned with length of the interval, then equation 1 is applied as below.

The table provides a rudimentary exemplification of applying equation 1 with some apocryphal values. The figurine below the table gives synonymic pictorial view of the table. We presume that such an approach would assist the reader in better understanding the intuitiveness behind the algorithm.

TABLE I: Calculation of spare capacity

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

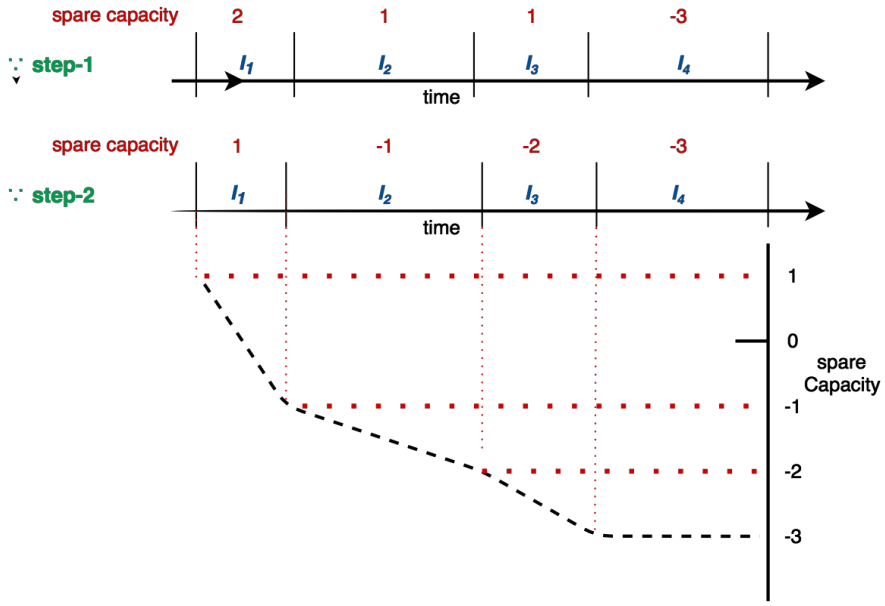

Fig. 2: Calculation of spare capacity

The example gives a clear picture of *ripple of negative intervals*. Graphical representation shows that I_1 is not the only lender interval, rather many intermediate intervals partially satisfy the computational need of I_4. Like many lenders the relation window might also have multiple borrowers and also combination of both.

The same approach of assurance is provided during online phase for accepted firm aperiodic tasks.

Interval progression. During online phase as time progresses the current interval's end is checked against current time and progressed to next interval when needed. This is represented algorithmically as function *update-slot*.

Spare Capacity update. For each slot the job is selected based on EDF. Selected job is checked for interval to which it belongs. If job belongs to the current interval, then the spare capacity of the interval is untouched as the needed computation is already done either during offline phase(in case of periodic tasks) or online phase(in case of accepted and guaranteed firm aperiodic task). If the task does not belong to the current interval the 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. If the job's interval is positive, it is simply added with one slot. This process is functionally represented in algorithm labelled *update-sc*. This functionality repeats itself for every slot.

1) Neutralisation Effect.: On execution of spare capacity update for every slot, an effect is observed when the current interval I_{curr} and the job $J_{i,k,curr}$ is within the same relation window,ϱ_i, and $I_{J_{i,k}}.\omega$ is negative.This effect is named neutralisation effect. The observation is as follows:

1 If the $J_{i,k,curr}$'s interval is not I_{curr}, then we first negate the I_{curr} by one slot.
2 Then we go to interval $I_{J_{i,k}}$, test for spare capacity negativity and add one slot to it.
3 Now we traverse backward to each consecutive negative interval and add one slot. Repeat this process till the first positive interval is noticed and also add one slot to it.The first positive interval in this case is I_{curr}, so we neutralise what we negated in the step 1.

This effect is important because it leads to a rule of application in deferred update, explained later in this paper called *neutralisation rule*.

Example: Exemplification of neutralisation effect[2]

Let's assume $I_{J_{i,k,curr}} = 4$ and $I_{curr} = I_1$, so both the current job and current interval is within the same relation window ϱ_i. The following is an elementary view of the interval's spare capacity before $J_{i,k,curr}$ is scheduled with apocryphal values.

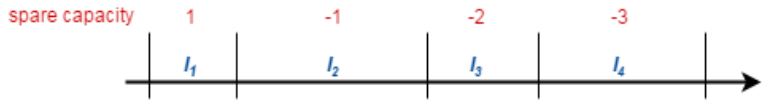

Fig 3-(a): Initial state before $J_{i,k,curr}$ is scheduled.

Fig 3-(b): On decision 1^{st} the job's interval nativity is checked. In this scenario, the job does not belong to the current interval, so one slot is negated from the I_{curr}, i.e., from I_1, so the I_1's spare capacity becomes 0.

Fig 3-(c): Next the job's interval,I_4 is tested for negativity. In this scenario it is negative, i.e., "-3". Irrespective of negativity, the job's interval is incremented by one, making the interval $I_4 = -2$.

Fig 3-(d): Since the job's interval is tested positive in negativity check, all the consecutive back intervals including the 1^st positive interval is incremented by one, i.e., intervals I_3, I_2 and I_1 is incremented by one.

Fig. 3: Neutralisation effect

If noticed interval I_1 is negated and added with same value at same point of time, causing neutralisation effect on I_{curr}, i.e., I_1.

[2]Illustration provided is from interval perspective, because the paper tries to reason the problems from an abstract interval viewpoint.

Algorithm 1: Slotshifting Online Phase

Function schedule($J_{i,k,Curr}$);
Input : $J_{i,k,Curr}$: job previously selected
Output: $J_{i,k,nxt}$: next selected job
update-sc($J_{i,k,Curr}, I$) ▷ update the interval;
if $A \neq \emptyset$ **then**
  Test-aperiodic(ι_A) ▷ Test and try Guaranteeing firm Aperiodic's if any;
end
$J_{i,k,nxt} \leftarrow$selection-fn() ▷ Get next eligible job;
update-slot();
return $J_{i,k,nxt}$;

Function update-sc($J_{i,k,Curr}, I$);
Input : $J_{i,k,Curr}$: previous selected job, I: Interval list
Output: $\emptyset$
Result: Spare Capacity of the respective intervals get updated
if $J_{i,k,Curr}.I == I_{curr}$ **then**
  return;
end
$I_{curr} \leftarrow I_{curr} - 1$;
$I_{tmp} \leftarrow J_{i,k,Curr}.I$;
while I_{tmp} **do**
  if $I_{tmp}.\omega \geq 0$ **then**
    $I_{tmp}.\omega \leftarrow I_{tmp}.\omega + 1$;
    break;
  end
  else
    $I_{tmp}.\omega \leftarrow I_{tmp}.\omega + 1$;
    $I_{tmp}.\omega \leftarrow I_{tmp}.prev$;
  end
end

Function update-slot();
Input : $\emptyset$
Output: $\emptyset$
Result: s_{cnt} gets added by 1; also check and move I_{curr} to $I.nxt$
$s_{cnt} \leftarrow s_{cnt} + 1$;
if $s_{cnt} \geq I_{curr}.end$ **then**
  $I_{curr} \leftarrow I_{curr}.nxt$;
end

Function Test-aperiodic(ι_A, I);
Input : ι_A: Unconcluded firm aperiodic list;
I: list of intervals
Output: $\emptyset$
Result: Job in ι_A is moved to either ι_G or ι_S
while $\iota_A \neq None$ **do**
  $\Gamma \leftarrow \iota_A.dequeue()$;
  if $I_{acc} \leftarrow$ *Acceptance-test*$(\Gamma, I) \neq None$ **then**
    Guarantee-job(I, I_{acc}, Γ);
    $\iota_G.queue(\Gamma)$
  end
  else
    $\iota_S.queue(\Gamma)$
  end
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.end ≤ 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 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
17 if I_acc.end > Γ.C then
18 |  I_New = split-intr(I_acc, Γ.C);
19 end
20  I_iter ← I_acc;
21 while I_iter.id ≥ I_curr.id do
22 |  J_lst ← I_iter.J_lst;
23 |  ω ← 0;
24 |  J ← J_lst.head;
25 |  while J do
26 |  |  ω ← ω + J.c;
27 |  |  J ← J.nxt;
28 |  end
29 |  I_iter.ω ← ω + max(0, I_iter.nxt.ω);
30 |  I_iter ← I_iter.prev;
31 end
```

III. Problem Statement

In real-time systems, scheduling algorithms are required to improve the use of the computational resources of the system. Thus, these scheduling algorithms must feature low overheads to leverage as much computational resources as possible for the applications at runtime. On the one hand, they must provide a satisfactory level of predictability and worst case guarantees for the tasks. On the other hand, they are expected to provide flexibility to react to aperiodic tasks. Last but not least, it is desirable that these algorithms must be able to handle the more and more complex constraints between interacting applications. In this section we will provide a detailed overview of the overheads in the current implementation of slot shifting.

• ***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. If the mentioned is the

case, we need to 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.

The function update-sc becomes an consistent Overhead when:

* current executing job's Interval is not current interval and the job interval's spare capacity is a big negative number. The big negative spare capacity intuitively means the borrow from the lender interval is 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 would be a normal scenario in slot level EDF. In other words, once the backward spare capacity update procedure is started between intervals due to negative job's interval's spare capacity, the probability of repeating for next consecutive slots till the job's interval's spare capacity becomes positive is almost 1.

• ***Problem2:*** *Slots*

Although there exists systems, such as avionic systems[29] that are based on the notion of slots, most of the existing real-time systems do not use slot. The notion of slots adds following overhead on system:

* Increases the number of time scheduling decision needs to be made.
* The worst case execution time of the task is calculated based on

$$C_i = \left\lceil \frac{calculated\ upper\ bound\ of\ Job\ execution\ time}{slotlength} \right\rceil \tag{2}$$

In the above defined equation, we notice the *approximation of computation time* of the jobs to its ceiling to fit the notion of slot. There might be some cases where the rounding of computation might take off more time from the system than needed by the tasks. This approximation could be part of the time needed to accept a firm aperiodic tasks during online phase, causing admittance failure.

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

When an aperiodic arrives and gets accepted then it needs to be guaranteed. As described, guarantee algorithm creates an interval if needed and applies equation (1) starting from the last interval in which job was accepted till the current interval. 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:

* **Needs More data space.** Both job and interval needs to hold reference[3] of each other, needing an unnecessary additional place holder in either interval or job.
* **Implementation complexity.** Adds additional difficulty in implementation, because along with the traversal of intervals, we also need to traverse through the jobs per interval.
* **More latency.** Increases runtime complexity due to application of equation 1 to calculate spare capacity for each interval.
* **Reduces predictability.** Calculation of spare capacity on each interval would take different runtime. The calculation runtime per interval 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.

IV. Proposed Solution

In this section we define a new algorithm inspired from slot shifting but with the mentioned problems addressed. 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.

As explained in Section 3 problem 1, the function that updates the spare capacity every slot has considerable overhead. The complexity of spare capacity update is $O(n)$ per slot, where n is the computational complexity involved to traverse the intervals backwards till the first positive interval. The whole update procedure is a redundant repeatable procedure that occurs for each slot, this aggregates the complexity of spare capacity update for an interval to $O(n * i)$, where i is computational complexity to update the spare capacity every slot within the interval.

On observation, we infer that accumulating i slots spare capacity update procedure and applying them later (when necessary, i.e., either at end of the interval or on aperiodic task arrival) would curtail the redundancy in update

procedure and thus reducing computation to $O(n)$ per interval from $O(n * i)$. This procedure of accumulating and updating the spare capacity on need basis is termed as **deferred update**.

Offline and preparation phase. To make this deferred update work, offline phase is complicated with an additional computation that would interweave the references of the relation window, i.e., if the relation window, ϱ exists between a set of intervals, then each interval in ϱ's lender, l and lent-till, b field is delegated with the respective reference of lender and lent-till[3].

1) O1 update spare capacity: Additionally, every Interval is added with an additional field named *update-val*, u. The update-val is updated during online phase through algorithm termed as **O1-Update-sc**, described below.

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

/***neutralisation rule** 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 $J_{i,k,curr}.I.l == I_{curr}.l$ *and* $tsk_\omega < 0$ **then**
  return;
end
$I_{curr}.\omega \leftarrow I_{curr}.\omega - 1$;

Described O1-Update-sc will get triggered at the end or beginning of every slot. When noticed the algorithm has no loops, but just checks on three conditions to update spare capacity and update-val of job's interval or just current interval's spare capacity. The verbose explanation of algorithm is as follows:

- Check the spare capacity of job's interval,ω_j is negative before updating, if yes, then along with ω_j update also job's interval's update-val, U_j.
- *Neutralisation rule.* 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 $O(1)$.

2) Naive Deferred update: 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 to guarantee.

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

[3]References are method to directly access the specified object without any traversal or computational overhead. Similar to ANSI C's pointer notion.

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 $lentTill \neq I_{curr}$ **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 1 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 $b_i.\omega$.

Symbolizing the functions. 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. The ω_i represents the spare capacity of the interval i, the subscript,i that represent a interval is made optional, rather the explanation in examples holds a column representing the corresponding interval id. We would Symbolize 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 $\curvearrowright^N$.

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 is the only lender. This is generally not the case; There could be an interval, $I_{\varrho_i,btw}$ in the middle of 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, we will first derive the offline version of spare capacity calculation in 2 steps.

TABLE II: 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 $I_{\varrho_i,btw}$, precisely, I_2 and I_3, partially satisfies capacity constraint of $I_{\varrho_i,aft}$, i.e., I_4. Assuming deferred update offline phase is applied on this set of intervals, we label this relation window as ϱ_1. In this exemplification, let's assume I_{curr} starts at I_1 and I_4 is the last interval. During online phase for spare capacity maintenance Algorithm(2) is applied on each slot and Algorithm(3) is applied when necessary, i.e., either when aperiodic arrives or end of interval. The table below shows slot by slot update procedure.

TABLE III: 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 III.

TABLE IV: 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**, due to an interval, $I_{\varrho_i,btw}$ becoming positive during deferred update. The value above 0 of the interval,$I_{\varrho_i,btw}$ in relation window should not be propagated to back intervals, or the value after the interval,$I_{\varrho_i,btw}$ becoming positive should be treated as an error to backward interval.

Scenario 2: *During online phase, middle of ϱ_i some interval becomes positive before deferred update.* 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 V: 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 VI: 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}, The I_{curr} should be negated by one, because I_2 has already become positive so

neutralisation effect should stop at I_2. The problem is due to the fact that I_3 is not aware 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.

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

4) Formalizing Solution: To correct the error we now define ε in 2 steps each rectifying error due to the above mentioned scenarios and we would conglomerate them into one single solution. This conglomerated solution would be applied on naive-deferred-update to make it into a complete working deferred spare capacity update solution.

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 after intervals traverses backward. 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 probe the above formulation closer into equation(3) and equation(4), both are the variant of the same formulae applied iteratively during $\curvearrowright \varrho_i$. The REC in this scenario is just the summation of all $\omega_i \geq 0$.

Addressing scenario 2. To understand the solution, lets understand the cause for the error. The error is due to assumption that during online phase $I_{\varrho_i,btw}$ will not become positive before deferred update is applied, but this assumption is wrong. An interval in between,$I_{\varrho_i,btw}$ can become positive satisfying the borrowed values of $I_{\varrho_i,aft}$ before deferred update, due to some job $J_{i,k,curr}$ belonging $I_{\varrho_i,btw}$ runs satisfying the constraints. Intuitiveness of solution is that, spare capacity of the intervals $I_{\varrho_i,aft}$ that executes in I_{curr} should not propagate its U beyond $I_{\varrho i,btw}$. To make this work, U accumulated after $I_{\varrho_i,btw}$ should be made into an error to $I_{\varrho_i,bfr}$. This inference leads to

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

.

- $c \leftarrow$ *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.
- $\omega_{c,bfr}$ *is the spare capacity of the interval before deferred update on the interval* c.

In words, The interval that is currently getting updated is first checked weather its spare capacity is greater than or equal to zero, if yes, then the update-val from the after intervals,$I_{\varrho_i,aft}$ becomes the current REC val, ϵ_c, else proceed with equation 3 and 4. If the spare capacity of the interval that is getting updated,$I_{\varrho_i,btw+x}$ is found positive before update, then REC value collected previously is applied on the updating interval, then the update val from all the after interval becomes the REC for the before intervals,$I_{\varrho_i,bfr}$.

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.

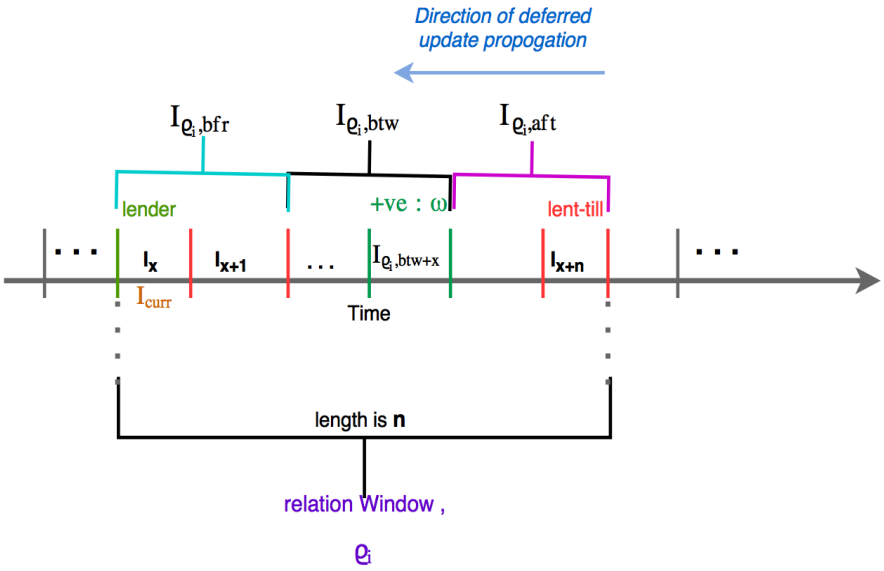

Fig. 4: **Interval between the ϱ_i becoming positive before deferred update applied.**

5) Complete working 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 intervals in 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 $lentTill_{temp} \neq I_{curr}$ **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* $lentTill_{temp} \neq I_{curr}.b$ **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

The previous section, deferred update of spare capacity almost removed the complexity involved due to spare capacity update per slot, by deferring it. In this section we would remove the notion of slot, making the slot level spare capacity shifting into a time based capacity shifting. 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 in slot shifting gets triggered for every slot, here on needs to be triggered at the end of each interval.i.e.,

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

  The timer expiry is no more constant as it was in slot shifting.
- we need to alter the slot based O1-Update-SC into capacity based spare capacity update as mentioned in Algorithm 4.

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: $\emptyset$
Result: Checks and Updates only spare capacity of job and current interval

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

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

Notion in words, 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 update-val of the current job, $u_{J_{i,k,curr}}$ 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(Algorithm 4) **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 $Timer$, λ 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

In this section we derive a new mode of guaranteeing the firm aperiodic task, which would take constant time per interval and would stop traversal once the computation constraint,$\Gamma.c$ for the aperiodic task is satisfied, thus solving the problems mentioned in section3 problem3.

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: ∅
   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 words, the algorithm uses a approach of delta, Δ which will initially hold the $\Gamma.c$ and updates the intervals till the Δ becomes 0. Negation of Δ happens in the below mentioned steps, which take place 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.

- 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.

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.

D. Putting it all together

This section tries to show how the capacity shifting algorithm will incorporate into the online phase. The algorithm removes the notion of slot, s and slot count s_{cnt}.

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 $\iota_A \neq \emptyset$ **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;
$J_{i,k,nxt} \leftarrow$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: $\emptyset$
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;
  $J_{i,k,curr}.interval.\omega \leftarrow J_{i,k,curr}.interval.\omega + run - time$;
  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
  $J_{i,k,curr}.interval.\omega \leftarrow J_{i,k,curr}.interval.\omega + run - time$;
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
$I_{curr}.\omega \leftarrow I_{curr}.\omega - run - time$;
Function update-time();
Input : $\emptyset$
Output: $\emptyset$
Result: checks and moves I_{curr} to $I_{curr}.nxt$ along with deferred-update
if *get-curr-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: $\emptyset$
Result: Job in ι_A is moved to either ι_G or ι_S
while $\iota_A \neq \emptyset$ **do**
  $\Gamma \leftarrow A.dequeue()$;
  if $I_{acc} \leftarrow$ *Acceptance-test*$(\Gamma, I) \neq \emptyset$ **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 $lentTill_{temp} \quad \neq I_{curr}$ **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* $lentTill_{temp} \quad \neq I_{curr}.b$ **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

```
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 ∅
I_tmp ← I_curr;
ω_tmp ← 0;
while I_tmp ≤ Γ.d do
    if I_tmp.ω > 0 then
        ω_tmp ← I_tmp.ω + ω_tmp;
    end
    I_tmp ← I_tmp.nxt;
end
if ω_tmp ≥ Γ.C then
    return I_tmp.prev;
end
else
    return ∅;
end

/* New Version of guarantee algorithm;
Notice that algorithm has only one loop and stops at a point Δ satisfies;
*/;
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: ∅
  Result: Γ is guaranteed
if I_acc.end > Γ.C then
    I_New = split-intr(I_acc, Γ.C);
end

if I_New then
    I_iter ← I_New;
end
else
    I_iter ← I_acc
end
Δ ← Γ.C;
while Δ do
    if I_iter.ω < 0 then
        I_iter.ω ← I_iter.ω − Δ
    end
    else if I_iter.ω > 0 then
        if I_iter.ω ≥ Δ then
            I_iter.ω ← I_iter.ω − Δ;
            Δ ← 0;
        end
        else
            I_iter.ω ← I_iter.ω − Δ;
            Δ ← Δ − I_iter.ω;
        end
    end
    I_iter ← I_iter.prev;
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 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 we haven't made it part of this algorithm just to keep the gist of the algorithm simple and understandable.

When the whole algorithm is observed, the complexity of scheduler within interval due to the 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. This was the complexity of the traditional algorithm per slot, which is now reduced to per interval.

Guarantee-job. In slot shifting algorithm the 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. The proposed algorithm reduced this complexity to $O(1)$, moreover backward traversal is cut short from "I_acc to I_{curr}", to "I_{acc} to a point where computational constraint of the aperiodic is satisfied".

V. Experimental Results

Above, we proposed a new version of slot shifting scheduling algorithm and showed that this algorithm often makes better use of available computing resources than pure slot shifting algorithm. We now experimentally evaluate Algorithm proposed algorithm and compare its performance with that of slot shifting.

we have made use of the pseudo random task set generators by Ripoll et al. [11] and UUnifast [12] for evaluating the feasibility analysis of the algorithm. This was made available in Simso framework. Workloads generated by mentioned generator have been widely used for experimentally evaluating real-time scheduling algorithms, and these experiments have been revealed to the larger research community for several years now. We believe that using this task generator provides a context for our simulation results, and allows them to be compared with other results performed by other researchers.

UUnifast generator: It generates the taskset configurations with a fixed number of tasks (N), specified value of sum of their individual utilization factor (U) and number of taskets to generate (nset). UUniFast produces independent tasks with randomly generated unbiased utilization factors. In this process, the utilization factor of each task is sampled and then only the tasks with correct total utilization are kept in the configurations. The task count is drawn between [5; 20] and utilisation factor is restricted between [10; 90] percent. The nset is kept constant count of 1.

Ripoll generator: It generates taskset configuration based on number of tasksets to generate (nsets), maximum computation time of a task (Compute), maximum slack time (deadline), maximum delay after deadline (period) and total utilisation each set should reach (target-util). The first three parameters are used to generate each task of the task set. The processor utilization, is used to determine the number of tasks. The the utilization factor of the system which is uniformly drawn from interval [10; 90], the computation times are uniformly chosen from the interval [1; 20], the deadlines from the interval [2; 170], and the periods from the interval [3; 650].

Further the interval range of the input paramter is further randomized by simple random generator. The results of the generator is passed as input to offline phase of the slot shifting to generate job and interval table from the task-set to make it accustomed for slot shifting algorithm. The same framework is used for aperiodic taskset, but early start time is randomized further between the range of the hyper period.

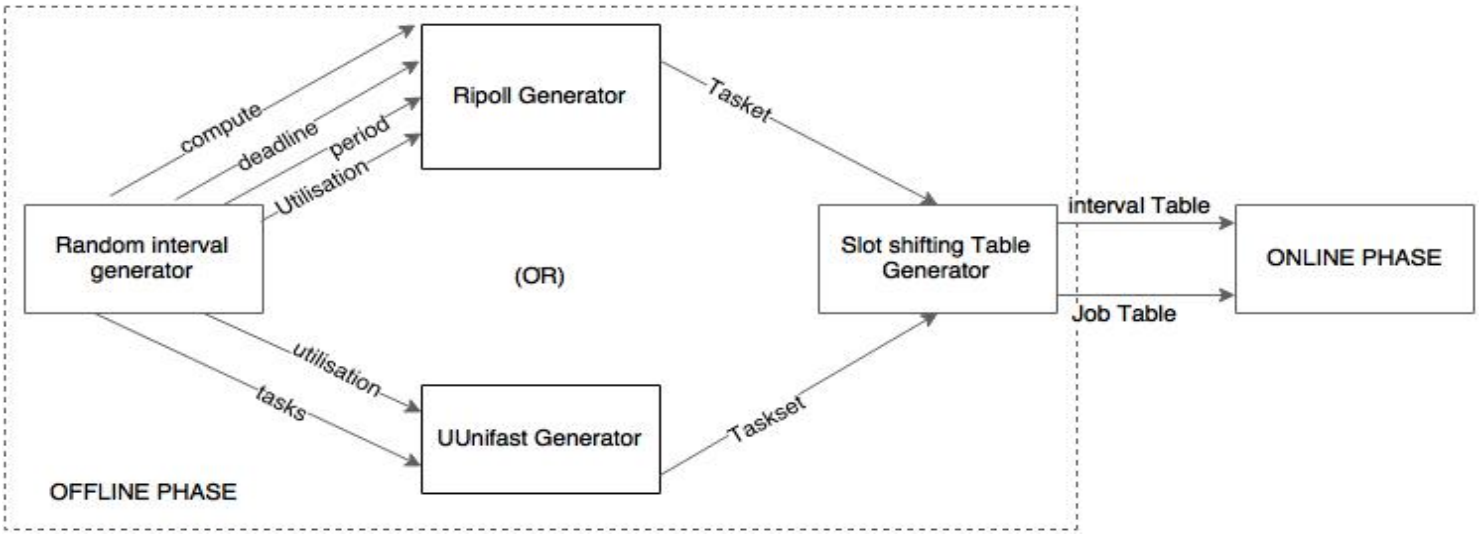

Fig. 5: Offline Task Generator and Slot shifting offline table generator framework.

To give the right benchmark, online and offline phase of both existing Slot shifting and proposed algorithm was implemented in SimSo [18, 19] simulation framework. To make the whole implementation adaptable to changes with respective slot shifting; A generic object oriented framework was created within SimSo to adapt any future changes with respect to Slot shifting.

Maximum of upto 5000 jobs are used to test the correctness and benchmark between traditional and new algorithm's computational need. The slot length was made constant of 5 Milliseconds in slot shifting.

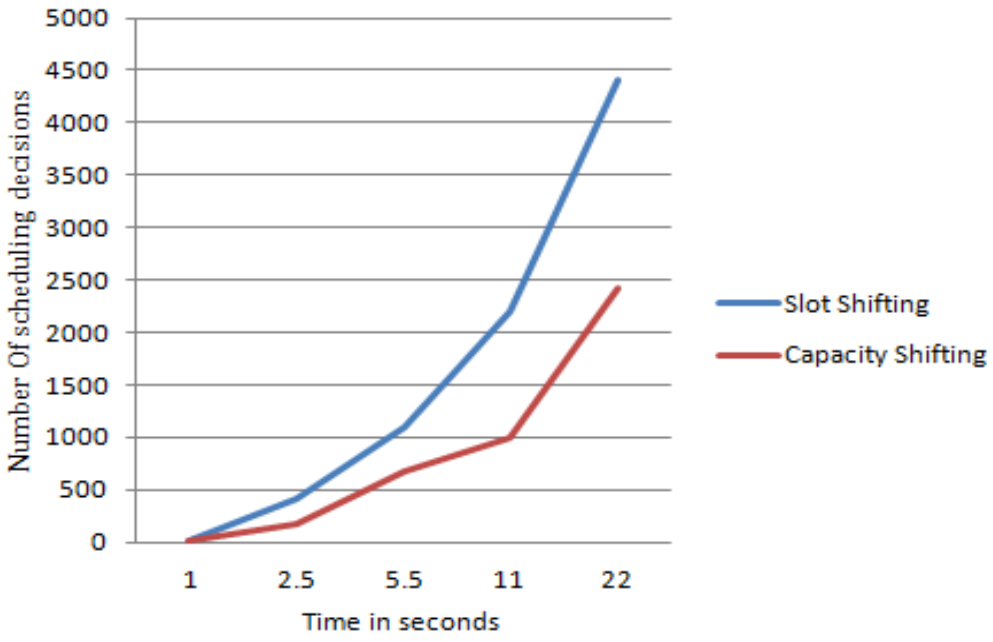

Fig. 6: Performance improvement in number of scheduling decisions with Capacity shifting against slot shifting.

The vertical axis shows the number of scheduling decisions taken on an average 2604 jobs with 4400 slots decision in slot shifting against capacity shifting. We can infer that capacity shifting takes on an average of 45 percent less scheduling decision against the slot shifting. In some best cases the improvement is seen going upto 60 percentage.

Guarantee Algorithm. To benchmark the guarantee algorithms of slot shifting against capacity shifting, we exploited the feature in SimSo simulator of rounding or setting up computation time per functionality. To make the benchmark visible we scaled the time to iterate one job within the interval to apply equation(1) during guaranteeing process to 1 nanosecond and considered other miscellaneous computations as 0.5 nanosecond.

Fig 7 below shows the constant time of computation in new guarantee algorithm which is as minimal as miscellaneous computation of the traditional slot shifting algorithm. Conceptually, the new guarantee algorithm computes spare capacity of the intervals with minimal constant computation complexity compared with slot shifting. The computation time of the algorithm is not effected by number of jobs the interval owns.

The vertical axis in Figurine 7 shows the computational time per interval and horizontal axis shows number of jobs in the interval. we can infer a constant computation time per interval of 0.5 nanosecond in new guarantee algorithm against a proportional increase in computation time as number of jobs increases in slot shifting algorithm.

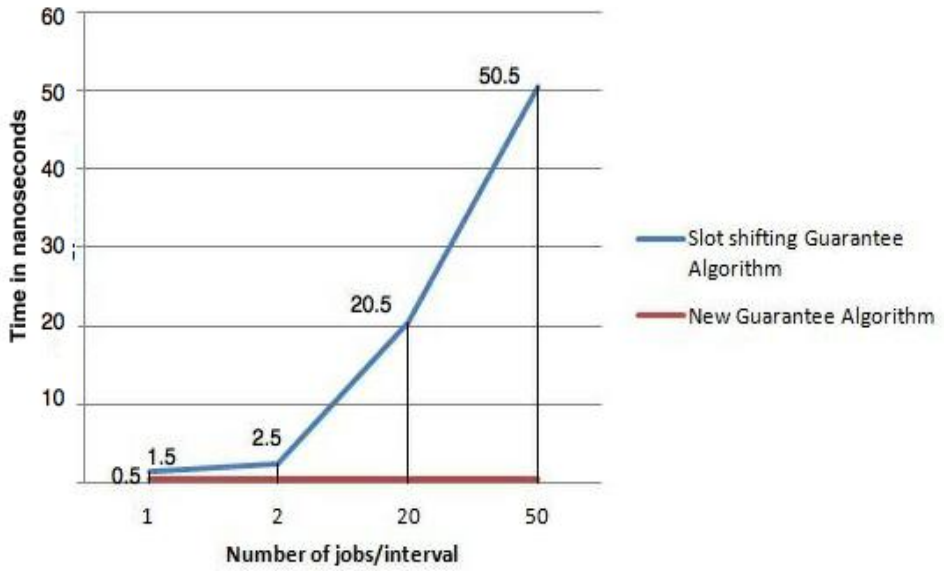

Fig. 7: Constant time taken by new Guarantee algorithm against slot-shifting Guarantee algorithm.

VI. Conclusion and future work

This paper presented 3 novel solutions, namely deferred update, slotless notion and O(1) guarantee algorithm. The solution 1 and 3 can be adapted to any other scenarios independently, but solution 2 is dependent on solution 1, in other words, solution 2 is an extension of solution 1. conglomerating all the 3 solutions is named capacity shifting.

Moreover following are some more areas of observation and areas to explore with respect to improvising Slot shifting or current algorithm capacity shifting.

- **Compute when required.** We tried to defer the computation of spare capacity to the end and beginning of intervals, but this notion could further be extended to an approach which does update of spare capacity when required.
- The capacity shifting could be extended to **multi core platforms**.
- The presented algorithm on deferred spare capacity update happens even if the update is not necessary. This could be avoided by small conditional checks in implementation.
- The remaining challenges:
  - A small drift in system clock against computed slot clock will cause the whole system failure.
  - The slot shifting was implemented in realtime platforms like $LITMUS^{RT}$ and in simulation environment like SimSo. The major problem was injection and preparation of precomputed tables of jobs and intervals into the system, i.e., the slot shifting needs both intervals and jobs be aware of each other. Creating reference of each other(jobs and intervals) for offline table is a strenuous effort. Getting such a data into the system needs separate preparation software. The effort to implement both offline and online phase of the preparation software is an effort equal to the implementation of the slot shifting algorithm itself.
  - Offline table can become really big. In some cases of non-harmonic task set of 10 tasks produced interval tables in count of 1000's for a hyper period, which we believe is normal in real environments.

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 deviating ideas of implementation and providing me with valuable suggestions on need. Finally I would like to thank my professor Gerhard Fohler for deriving a novel algorithm called Slot shifting and giving me this opportunity to implement a new version of it.

References

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

[2] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In Proceedings of the IEEE Real-Time Systems Symposium, December 1987

[3] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhancing aperiodic responsiveness in hard-real-time environments. IEEE Transactions on Computers, 4(1), January 1995.

[4] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In Proceedings of the IEEE Real-Time Systems Symposium, December 1992.

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

[6] 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.

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

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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

[16] 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.
[17] 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.
[18] 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
[19] Simso simulator framework official website.
`http://projects.laas.fr/simso`
[20] 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
[21] M. Spuri and G. C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In Proceedings of the IEEE Real-Time Systems Symposium, December 1994.
[22] M. Spuri and G. C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. Journal of Real-Time Systems, 10(2), 1996.
[23] 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.
[24] 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.
[25] Damir Isovic. Flexible Scheduling for Media Processing in Resource Constrained Real-Time Systems. PhD thesis, Malardalen University, Sweden, November 2004.
[26] 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.
[27] 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.
[28] 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.
[29] Aeronautical Radio, Incorporated (ARINC), https://en.wikipedia.org/wiki/ARINC