Determining Actual Response Time in P-FRP[†]

Chaitanya Belwal, Albert M.K. Cheng

Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

UH-CS-10-05
June 28, 2010

Keywords: Response Time, Schedulability Analysis, Real-time System, Functional Programming

Abstract

Functional Reactive Programming (FRP) is a declarative approach to modeling and building reactive systems. FRP has been shown to be an expressive formalism for building graphics, robotic, and vision applications. Recently, priority-based FRP (P-FRP) was introduced as a formalism that guarantees real-time response. Unlike the classical preemptive model[‡] of real-time systems, preempted events in P-FRP are aborted and have to restart when higher priority events have completed. The abort and eventual re-start makes the response time of a lower priority event completely dependent on the execution pattern of higher priority events. Though methods to determine approximate values for the response time of events in P-FRP have been presented, no convenient method has yet been established that can determine actual response time. A common method for computing such response time in the preemptive execution model is not guaranteed to give correct results in P-FRP. An obvious approach in P-FRP is running a time-accurate simulation. However, this approach is computationally expensive and not feasible in most practical situations. We show that an exhaustive enumeration technique for idle periods, is a more efficient technique than time accurate simulation, and can be easily adopted to determine actual response time in P-FRP and other transaction based execution models.

[†] This work is supported in part by U.S. National Science Foundation under Award no. 0720856

[‡] In this paper the classical preemptive model refers to a real-time system in which tasks can be preempted by higher priority tasks, and can resume execution from the point they were preempted

Determining Actual Response Time in P-FRP[*]

Chaitanya Belwal and Albert M.K. Cheng
Department of Computer Science
University of Houston, TX, USA

Abstract

Functional Reactive Programming (FRP) is a declarative approach to modeling and building reactive systems. FRP has been shown to be an expressive formalism for building graphics, robotic, and vision applications. Recently, priority-based FRP (P-FRP) was introduced as a formalism that guarantees real-time response. Unlike the classical preemptive model[§] of real-time systems, preempted events in P-FRP are aborted and have to restart when higher priority events have completed. The abort and eventual restart makes the response time of a lower priority event completely dependent on the execution pattern of higher priority events. Though methods to determine approximate values for the response time of events in P-FRP have been presented, no convenient method has yet been established that can determine actual response time. A common method for computing such response time in the preemptive execution model is not guaranteed to give correct results in P-FRP. An obvious approach in P-FRP is running a time-accurate simulation. However, this approach is computationally expensive and not feasible in most practical situations. We show that an exhaustive enumeration technique for idle periods, is a more efficient technique than time accurate simulation, and can be easily adopted to determine actual response time in P-FRP and other transaction based execution models.

Index Terms

Response Time, Schedulability Analysis, Real-time System, Functional Programming

I. Introduction

Functional Reactive Programming (FRP) [26] is a declarative programming language for modeling and implementing reactive systems. It has been used for a wide range of applications, notably, graphics [8], robotics [19], and vision [20]. FRP elegantly captures continuous and discrete aspects of a hybrid system using the notions of *behavior* and *event*, respectively. Because this language is developed as an embedded language in Haskell [12], it benefits from the wealth of abstractions provided in this language. Unfortunately, Haskell provides no real-time guarantees, and therefore, neither does FRP.

To address this limitation, resource-bounded variants of FRP were studied ([16],[24],[25]). Recently, it was shown that a variant called priority-based FRP (P-FRP) [16], combines both the semantic properties for FRP, guarantees resource boundedness, and supports assigning different priorities to different events. In P-FRP, higher priority events can preempt lower-priority ones. However, to maintain guarantees of type safety and stateless execution, the functional programming paradigm requires the execution of a function to be atomic in nature. To comply with this requirement, as well as allow preemption of lower priority events, P-FRP implements a transactional model of execution. Using only a copy of the state during event execution and atomically committing these changes at the end of the event handler, P-FRP ensures that handling an event is an "all or nothing" proposition. This preserves the easily understandable semantics of the FRP and provides a programming model where response times to different events can be tweaked by the programmer, without ever affecting the semantic soundness of the program. Thus, a clear separation between the semantics of the program and the responsiveness of the implementation of each handler is achieved.

[*] This work is supported in part by U.S. National Science Foundation under Award no. 0720856

[§] In this paper the classical preemptive model refers to a real-time system in which tasks can be preempted by higher priority tasks, and can resume execution from the point they were preempted

This transactional execution model used in P-FRP is not new, and several such models have been presented in the past. These are the transactional memory systems [14], lock-free execution for critical sections [1] and pre-emptable atomic regions in Java [18]. The development of these systems was primarily motivated by the need to avoid concurrency or precedence constraint issues, which have been a problem in the classical preemptive model [23]. In spite of its varying uses, the transactional model is not well understood. While several response time studies for this model are available ([1],[4],[9],[18]), they only provide basic schedulability analysis by modifying methods developed for the preemptive model. A study to find actual response time for this execution model has not been presented yet.

Several functional programming languages are being used in the industry, like for mission critical telecommunication equipment (Erlang [9]) and control of hybrid vehicles (Atom [13]). However, the temporal properties of software written in functional programming have not been well studied. With functional programming becoming a standard feature in popular software development platforms like Microsoft's Visual Studio [11], studies on its temporal and space properties are valuable. Previous work ([16],[21]) on P-FRP provided basic results on schedulability and approximate upper bounds on response times. Though approximate upper bounds only provide a general idea on the schedulability of events, the methods to compute them are much faster ([3],[7],[21]). In this paper we use the term 'actual' to differentiate from approximate or bounded response time. Actual response time gives an exact indication of the temporal properties of events in the system. Hence, actual response time is more useful when an accurate model of the system is required , such as in the design phase of a real-time system, or in developing exact schedulability tests.

An iterative method first presented by Audsley et al in [2] (termed Audsley's method in this paper), is a common approach for determining actual response time in the preemptive model. In this method, it is assumed that the amount of processor time taken by an event to execute, is constant and equal to its worst-case execution time (WCET). However, since a preempted event is aborted, the amount of processor time taken by a lower priority P-FRP event to complete execution, can be larger than its WCET, and thus not known *a priori*. Due to this reason the method in [2] is not guaranteed to work with P-FRP (see section III for example), and new methods for determining actual response time in P-FRP are required.

I.I Contributions

This paper presents an efficient algorithm that can be used in place of a simulation, to determine the actual response times of events in P-FRP. This is an essential step for making this technology practically usable since it is not practical to work out these response times by hand or simulation methods. To conform to terminology used in referenced real-time system papers, P-FRP events will be referred to as tasks in the rest of this paper. The utility of this work also extends to determining actual response time in systems which have similar execution models ([1],[4], [9], [18]).

After reviewing basic concepts and the P-FRP execution model (Section II) we:

- Present Audsley's iterative method for computing actual response time in the preemptive model (Section III)
- Introduce the time-accurate algorithm that simulates the execution model of P-FRP (Section IV)
- Present an enumeration technique for idle periods, which for the sake of brevity has been termed the *gap-enumeration* method (Section V)
- Present an algorithm that determines the actual response time of a task using the gap-enumeration method (Section VI)
- Provide performance analysis between the time accurate and gap-enumeration algorithms (Section VII)

And conclude by reviewing related work (Section VIII) and a reflection on these results (Section IX).

II. Basic Concepts and Execution Model of P-FRP

In this section, we introduce the basic concepts and the notation used to denote these concepts in the rest of the paper. In addition, we review the P-FRP execution model and assumptions made in this study.

II.I Basic Concepts

Essential concepts for P-FRP are tasks and their associated priority, their associated time period and the dual concept of arrival rate, and their processing time; the concept of a time interval and release offset therein. In our task model, all these assumed to be known *a priori*. The notation and formal definitions for these concepts as well as a few others used in the paper are as follows:

- Let **task set** $\Gamma_n = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of n periodic tasks
- The **priority** of $\tau_k \in \Gamma_n$ is the positive integer k, where a higher number implies higher priority
- T_k is the **arrival time period** between two successive jobs of τ_k and $r_k = 1 / T_i$ is the **arrival rate** of τ_k
- C_k is the **worst-case execution time** for τ_k
- $t_{copy}(k)$ is the time taken to make a copy of the state before τ_k starts execution (see section 2.2.1)
- $t_{restore}(k)$ is the time taken to commit the state after τ_k has completed execution (see section 2.2.1)
- P_k is the **processing time** for τ_k. Processing of a task includes execution as well as copy and restore operations. Hence, $P_k = t_{copy}(k) + C_k + t_{restore}(k)$
- $R_{k,m}$ represents the **release time** of the mth job of τ_k
- Φ_k represents the **release offset** which is the release time of the first job of τ_k. Or, $\Phi_k = R_{k,1}$. Hence, $R_{k,m} = \Phi_k + (m\text{-}1)\cdot T_k$
- A **level-*k* idle point** is a point in time, t in which no task having a priority of k or higher is awaiting execution and ready to execute strictly before t
- A finite contiguous interval of non-zero length $[t_1,t_2)$ is a ***k*-gap**, if every $t \in [t_1,t_2)$, is a level-$(k+1)$ idle point. The new term k-gap, denotes the time interval in which a task τ_k can be processed and is introduced for the sake of clarity
- The **threshold** of the k-gap $[t_1, t_2)$ is time t_1
- $\mathrm{T}\big|_{t_1}^{t_2}$ represent the **time window** for analyzing gaps, such that: $\forall t \in \mathrm{T}\big|_{t_1}^{t_2}$, $t_1 \le t < t_2 \wedge t_1 \ne t_2$. This new notation is used to differentiate from k-gap time intervals
- D_k is the **relative deadline** of τ_k. If some job of τ_k is released at time $R_{k,m}$ then τ_k should complete processing by time $R_{k,m} + D_k$, otherwise τ_k will have a **deadline miss.** In this paper, $D_k = T_k$
- A **gap set** $\sigma_k(\mathrm{T}\big|_{t_1}^{t_2})$ contains all the unique k-gaps present in the time interval $\mathrm{T}\big|_{t_1}^{t_2}$. The k-gaps present in $\sigma_k(\mathrm{T}\big|_{t_1}^{t_2})$ are also disjoint:

  for any two gaps $[t_{x1},t_{y1})$, $[t_{x2},t_{y2}) \in \sigma_k(\mathrm{T}\big|_{t_1}^{t_2})$, if $t \in [t_{x1},t_{y1})$ then $t \notin [t_{x2},t_{y2})$
- $|\sigma_k(\mathrm{T}\big|_{t_1}^{t_2})|$ represents the number of k-gaps present in $\sigma_k(\mathrm{T}\big|_{t_1}^{t_2})$
- The **gap-transformation function** $\lambda(\sigma_k(\mathrm{T}\big|_{t_1}^{t_2}), \Gamma_n)$ takes as input the gap set σ_k, and task set Γ_n. The function returns the gap set of the next lower priority task:

  $$\sigma_{k\text{-}1}(\mathrm{T}\big|_{t_1}^{t_2}) = \lambda(\sigma_k(\mathrm{T}\big|_{t_1}^{t_2}))$$
- The **gap-search function** $\mu(\sigma_k(\mathrm{T}\big|_{t_1}^{t_2}), P_k)$ takes as input the gap set $\sigma_k(\mathrm{T}\big|_{t_1}^{t_2})$ and P_k, and returns the earliest k-gap larger than or equal to P_k present in σ_k:

  $$[t_{x1},t_{y1}) = \mu(\sigma_k(\mathrm{T}\big|_{t_1}^{t_2}), P_k), \text{ such that:}$$

  $$t_{y1} - t_{x1} \ge P_k \wedge \nexists\ [t_x,t_y) \in \sigma_k(\mathrm{T}\big|_{t_1}^{t_2}) \wedge t_y - t_x > P_k \wedge t_x < t_{x1}$$

  If the gap search function returns a k-gap with threshold less than 0, then a k-gap larger than P_k does not exist in $\sigma_k(\mathrm{T}\big|_{t_1}^{t_2})$
- The **computational steps** of an algorithm is a numerical measurement of the number of times major iterations inside the algorithm have been performed during execution. This value gives us a general idea of the performance of the algorithms considers in this paper

- The **start time** of a task is the time when the task starts processing for the first time after the release of its job. The start time of a job of the highest priority task will always equal to the time of its release
- The **response time** of a τ_k written as RT_k is the relative time after its release in which τ_k completes processing
- **Interference** on τ_k is the action where the processing of τ_k is interrupted by the release of a higher priority task. In P-FRP, an interference forces τ_k to abort and re-process later

II.II Execution Model and Assumptions

In this study all tasks are assumed to execute in a uniprocessor system with no precedence constraints. When job of a higher priority task τ_i is released, it can immediately preempt an executing lower priority task, and changes made by the lower priority task are rolled back. The lower priority task will be restarted when the higher priority task has completed processing. Due to P-FRP's transactional nature of execution, all tasks are assumed to run without concurrency constraints. In the algorithms to derive the actual response time of a task τ_j, we have considered the release offset of τ_j to be 0.

When some task is released, it enters a processing queue Q which is arranged by priority order, such that all arriving higher priority tasks are moved to the head of the queue. The length of the queue is bounded, and no two instances of the same task can be present in the queue at the same time. This requires a task to complete processing before the release of its next job. To maintain this requirement we assume a *hard* real-time system with task deadline equal to the time period between jobs. Hence,

$$\forall \tau_k \in \Gamma_n,\ D_k = T_k$$

A task set is schedulable in some time interval, only if no task in the set has a deadline miss.

Once τ_i enters Q two situations are possible. If a task of lower priority than i is being processed, it will be immediately preempted and τ_i will start processing. If a task of higher priority than τ_i is being processed, then τ_i will wait in the Q and start processing only after the higher priority task has completed. An exception to the immediate preemption is made during *copy* and *restore* operations which is explained in the following paragraph.

II.II.I Copy and Restore Operations

In P-FRP, when a task starts processing it creates a 'scratch' state, which is a *copy* of the current state of the system. Changes made during the processing of this task are maintained inside such a state. When the task has completed, the 'scratch' state is *restored* into the final state in an atomic operation. Therefore, during the restoration and copy operations the task being processed cannot be preempted by higher priority tasks. If the task is preempted after copy but before the restore operation, the scratch state is simply discarded. The context-switch between tasks only involves a state copy operation for the task that will be commencing processing. The time taken for copy ($t_{copy}(k)$) and restore ($t_{restore}(k)$) operations of τ_k is part of the processing time of the task, P_k.

Our current methods do not yet account for situations where higher priority tasks cannot preempt lower priority tasks. Hence, for the methods presented in this paper, the values of $t_{copy}(k)$ and $t_{restore}(k)$ for all tasks are kept same and equal to a single discrete time unit of processing. Hence,

$$\forall k \in \Gamma_n,\ t_{copy}(k) = t_{restore}(k) = 1.$$

Such small values of $t_{copy}(k)$ and $t_{restore}(k)$ is reasonable as copy and restore operations are only a fraction of the worst-case execution time of the task. However, for better accuracy of results, in ongoing work we are developing methods where the values of $t_{restore}(k)$ and $t_{copy}(k)$ could be variable.

II.II.II Critical Instant in P-FRP

In response time analysis for fixed-priority scheduling, a *critical-instant* of release is assumed. Critical instant is the time, at which task releases lead to the worst-case response time (WCRT) [17] of the task being analyzed. In their seminal work, Liu and Layland [17] showed that in fixed-priority scheduling for the preemptive model, the critical-instant for a lower priority task τ_i occurs when it is released at the same time as the higher priority tasks. Or, the release offset of task τ_i and higher priority tasks is the same. This is also termed as a *synchronous* release of tasks. In P-FRP, a *synchronous* release of tasks does not lead to the WCRT, for all cases. Consider the following task set:

Task	*P*	*T*
τ_1	40	3
τ_2	12	4
τ_3	9	3

If all tasks are released synchronously, the response time of τ_1 is 24 (*Figure 1(a)*). However, if τ_2 is released at time 2 and τ_3 is released at time 5, the response time of τ_1 is 38 (*Figure 1(b)*). The response time of τ_1 is lower when higher priority tasks are released synchronously, showing that such a release does not lead to WCRT of a lower priority task.

The methods presented in this paper, determine the response time of a task only for a user specified release offset of higher priority tasks. Hence, the release offsets required by the methods presented in this paper, are assumed to be know *a priori*. This release offset, may or may not lead to the WCRT for the task being analyzed. To determine the WCRT for a given P-FRP task, all possible combinations of release offsets of higher priority tasks have to be generated. Then the time-accurate or gap-enumeration algorithms, presented in this paper, need to be used for computing the actual release time under each of the possible release offsets. Finally, the highest value of the response time computed for each release offset will be the WCRT for the task. In the example above, the WCRT of τ_1 is 38, computed by this method of evaluating all possible release scenarios.

III. Computing Actual Response Time in the Preemptive Model

In an important paper [2], Audsley et al demonstrated that if all tasks are synchronously released, the response time of τ_i (RT_i) can be determined using the following equation:

$$RT_i = P_i + B_i + \sum_{\forall j>i} \left\lceil \frac{RT_i}{T_j} \right\rceil \cdot P_j \qquad \ldots\ eq.\ 3.1$$

B_i is the blocking time due to concurrency control protocols, which is not applicable in our case. Since RT_i appears on both sides of the equation, an iterative approach using initial approximate values of RT_i can be used. If RT_i^n represents the nth approximate value of RT_i, and ignoring the blocking time, equation 3.1 can be written as:

$$RT_i^{n+1} = P_i + \sum_{\forall j>i} \left\lceil \frac{RT_i^n}{T_j} \right\rceil \cdot P_j \qquad \ldots\ eq.\ 3.2$$

The iteration starts with $RT_i^0 = 0$ and terminates when $RT_i^{n+1} = RT_i^n$. Since, in the preemptive model a synchronous release leads to the WCRT, equation 3.1 also computes the WCRT for τ_i.

Let's take a simple application of this equation, using the following P-FRP task set:

Task	*P*	*T*
τ_1	40	3
τ_2	12	4
τ_3	9	3

We have to compute the response time of τ_1 using equation 3.2, assuming a synchronous release of tasks:

Iteration 1, $n = 0$: $RT_1^1 = 3 + \left(\left\lceil \frac{0}{9} \right\rceil \cdot 3 + \left\lceil \frac{0}{12} \right\rceil \cdot 4 \right) = 3$

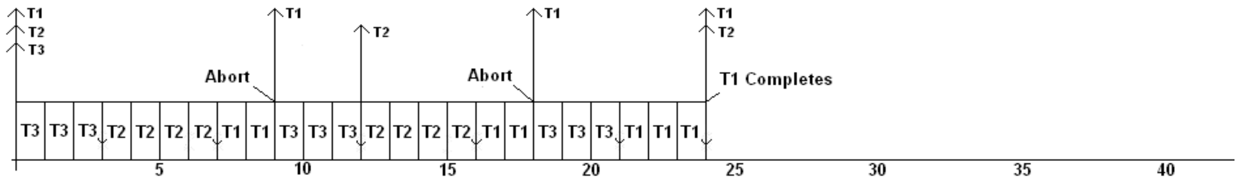
T1
T2
T3
Abort
T1 Completes
T3 T3 T3 T2 T2 T2 T2 T1 T1 T3 T3 T3 T2 T2 T2 T2 T1 T1 T3 T3 T3 T1 T1 T1
5
10
15
20
25
30
35
40

Figure 1(a): Task execution graph showing τ_1 completing processing at time 24 in a synchronous release. T1, T2 and T2 represent tasks τ_1, τ_2 and τ_3 respectively. The 'Abort' tag shows the time when τ_1 or τ_2 are aborted by higher priority tasks

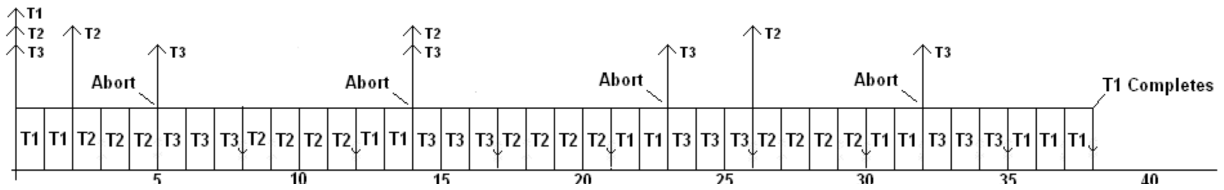
T1
T2
T3
Abort
T1 Completes
T1 T1 T2 T2 T2 T3 T3 T3 T2 T2 T2 T2 T1 T1 T3 T3 T3 T2 T2 T2 T2 T1 T1 T3 T3 T3 T2 T2 T2 T2 T1 T1 T3 T3 T3 T1 T1 T1
5
10
15
20
25
30
35
40

Figure 1(b): Task execution graph showing τ_1 completing processing at its worst-case response time of 38 when τ_2, τ_3 are released at times 2 and 5 respectively.

Iteration 2, $n = 1$: $RT_1^2 = 3 + (\left\lceil \frac{3}{9} \right\rceil \cdot 3 + \left\lceil \frac{3}{12} \right\rceil \cdot 4) = 10$

Iteration 3, $n = 2$: $RT_1^3 = 3 + (\left\lceil \frac{10}{9} \right\rceil \cdot 3 + \left\lceil \frac{10}{12} \right\rceil \cdot 4) = 13$

Iteration 4, $n = 3$: $RT_1^4 = 3 + (\left\lceil \frac{13}{9} \right\rceil \cdot 3 + \left\lceil \frac{13}{12} \right\rceil \cdot 4) = 17$

Iteration 5, $n = 4$: $RT_1^5 = 3 + (\left\lceil \frac{17}{9} \right\rceil \cdot 3 + \left\lceil \frac{17}{12} \right\rceil \cdot 4) = 17$

Since, $RT_1^4 = RT_1^5$, the iteration will terminate giving us the response time for τ_1 as 17. In *Figure 1(a)* we show the processing of tasks in the time window $\mathrm{T}|_0^{40}$, which shows the response time of τ_1 as 24. *Figure 1(a)* also illustrates the fact that, even though the processing time of τ_1 if 3 and is known *a priori*, τ_1 takes a total processor time of 7 to complete processing.

III.I Ras and Cheng's Modification for P-FRP

An attempt to apply Audsley's method in P-FRP was made by Ras and Cheng in [21]. An abort cost to the original equation has been added. The modified equation is given as:

$$WCRT_i = P_i + B_i + \sum_{\forall j \in hp_i} \left\lceil \frac{WCRT_i}{T_j} \right\rceil \cdot P_j + \sum_{\forall j \in hp_i} \left\lceil \frac{WCRT_i}{T_j} \right\rceil \cdot \max_{k=i}^{j-1} P_k \qquad \text{… eq. 3.3}$$

hp_i represents the set of tasks having a higher priority than τ_1. The initial value for $WCRT_i$ is set to P_i. This equation was proposed to compute the response time under a synchronous release. However, it could converge for only a few cases. Also, the authors' assertion that eq. 3.3 can compute the WCRT, is not quite correct. This is because a synchronous release does not always lead to WCRT in P-FRP. Applying equation 3.3 to our example, and setting $WCRT_1^0$ =20:

1: $WCRT_1^1 = 3 + (\left\lceil \frac{3}{9} \right\rceil \cdot 3 + \left\lceil \frac{3}{12} \right\rceil \cdot 4) + \left\lceil \frac{3}{9} \right\rceil \cdot 3 + \left\lceil \frac{3}{12} \right\rceil \cdot 4 = 17$

2: $WCRT_1^2 = 3 + (\left\lceil \frac{17}{9} \right\rceil \cdot 3 + \left\lceil \frac{17}{12} \right\rceil \cdot 4) + \left\lceil \frac{17}{9} \right\rceil \cdot 3 + \left\lceil \frac{17}{12} \right\rceil \cdot 4 = 31$

$$3:\ WCRT_1^3 = 3 + \left(\left\lceil \frac{31}{9} \right\rceil \cdot 3 + \left\lceil \frac{31}{12} \right\rceil \cdot 4\right) + \left\lceil \frac{31}{9} \right\rceil \cdot 3 + \left\lceil \frac{31}{12} \right\rceil \cdot 4 \quad = 51$$

$$4:\ WCRT_1^4 = 3 + \left(\left\lceil \frac{51}{9} \right\rceil \cdot 3 + \left\lceil \frac{51}{12} \right\rceil \cdot 4\right) + \left\lceil \frac{51}{9} \right\rceil \cdot 3 + \left\lceil \frac{51}{12} \right\rceil \cdot 4 \quad = 79$$

….

This computation will go on indefinitely and will never converge.

Clearly, Audsley's method, and its modified version are not guaranteed to compute the actual response time in P-FRP, and a different approach is required.

IV. Time-Accurate Simulation

A straightforward way to computing the response time in P-FRP, is to use a time-accurate simulation that progresses through every time tick and runs tasks based on the P-FRP execution model. The pseudo-code for such an algorithm is given in this section. The algorithm takes as input Γ_n and task τ_j, whose response time has to be ascertained. Between lines 4-32 a loop is executed, denoting every time step from *0* to T_j. In the loop between lines 4-15 tasks having higher priority than τ_j and which are released at that specific time tick, are added to the processing queue *Q*. In line 16 the length of *Q* is checked. If *Q* is empty then the highest priority task (τ_h) is extracted (lines 17-20), and the amount of time it has been processed for (e_h) incremented (line 22). The time for which other tasks have been processed is reset to 0 (lines 23 -25). This is because these tasks will have to restart after completion of the highest priority τ_h. If τ_h has completed processing, it is removed from *Q* (line 28).

If there is no higher priority task in *Q* it signifies a presence of a *j*-gap. At each consecutive tick for which there is a *j*-gap the size of variable *j*-gap is incremented (line 30) and when *j*-gap equals the processing time of τ_j, the time tick value is returned (line 31) denoting the response time of τ_j. If some higher priority task is processed before *j*-gap = P_j then the size of the *j*-gap is reset (line 21) to denote the restart of τ_j, at the next available *j*-gap.

```
1.  input: $\Gamma_n$, $\tau_j$
2.  output: response time of $\tau_j$
3.          -1 if task set is unschedulable
4.  loop $time \leftarrow 0, T_j$
5.      loop $\tau_i \leftarrow n$ to $j+1$
6.          $x \leftarrow \Phi_i$
7.          loop while ( $x \le time$ )
8.            if($x = time$)
9.              if($\tau_i$ present in Queue)
10.               return -1
11.             else
12.               Add $\tau_i$ to Queue
13.           $x \leftarrow x + T_i$
14.         end loop
15.     end loop
16.     if ($Queue\ Count > 0$)
17.       $h \leftarrow$ 1st task in Queue
18.       loop for every $\tau_i$ in Queue
19.           if ($h \le i$) $h \leftarrow i$
20.       end loop
21.       $j\text{-}gap \leftarrow 0$
22.       $e_h \leftarrow e_h + 1$
23.       loop for every $\tau_i$ in Queue
24.          if ($i$ != $h$) $e_h \leftarrow 0$
25.       end loop
26.       if($e_h = P_h$)
27.          $e_h \leftarrow 0$
28.          Remove $\tau_h$ from Queue
29.     else
30.       $j\text{-}gap \leftarrow j\text{-}gap + 1$
31.       if $j\text{-}gap = P_j$ return $time$
32.  end loop
```

IV.I Time Complexity

The time loop between lines 4-32 will execute for T_j time units in the worst case which is when τ_j will complete processing just before its next job. The loop at lines 5-15 iterates through $(n–j)$ tasks, hence lines 6-14 will run for maximum $(n–j)\cdot T_j$ times. The loop at lines 7-14 runs for every task, whose first job is released before the current time. Line 9 performs a search on the queue. The queue can contain maximum $(n\text{-}j)$ elements hence each execution of line 9 will take $(n–j)$ steps. If τ_k identifies the task with the highest arrival rate then line 9 will run for:

$$(1 + 2 \ldots .+ T_k) \cdot (n–j) = (n–j) \cdot T_k \cdot (1 + T_k)/2, \text{ which is bounded by } O((n–j) \cdot T_k^2).$$

Hence the execution steps of lines 8-13 are bounded by $O(T_j \cdot (n–j)^2 \cdot T_k^2)$. The loops at lines 18-20 and 23-25 execute for the maximum length of the queue. Hence the total number of steps that these loops can be executed for is bounded by $O((T_j – P_j) \cdot (n–j))$. The total worst case upper bound for this algorithm is:

$$O((T_j – P_j) \cdot (n–j)^2 \cdot T_k^2) + O((T_j – P_j) \cdot (n–j)) + O((T_j – P_j) \cdot (n–j)).$$

The dominating value is $O((T_j – P_j) \cdot (n–j)^2 \cdot T_k^2)$, which is the upper bound of this equation. τ_k is the task with the highest arrival rate.

Example: We ran the example given in section 3 through the time-accurate simulation algorithm, and computed the value of computational steps, which comes to 145.

V. The Gap-Enumeration Method

The time-accurate simulation method iterates through every time step till the response time of the task being analyzed is found. This approach is computationally intensive, since several iterations have to be performed. We present a different method using enumeration of *k*-gaps, based on the following characteristics of the P-FRP execution model.

Lemma 5.1: *A task τ_j can be processed only in elements of the set $\sigma_j(\mathrm{T}|_{t_1}^{t_2})$.*

Proof. The elements of $\sigma_j(\mathrm{T}|_{t_1}^{t_2})$ contain all the possible *j*-gaps. By its definition τ_j can only be processed in available *j*-gaps. Hence, τ_j can be processed in any of the elements of the set $\sigma_j(\mathrm{T}|_{t_1}^{t_2})$. □

Lemma 5.2: *For task τ_j to be schedulable, one j-gap of at least length P_j will exist between any two successive jobs of τ_j.*

Proof. To complete processing**,** τ_j should be processed for P_j time, without any interference from higher priority tasks. Since no task having a higher priority than *j* is available to be processed in a *j*-gap, τ_j will require a *j*-gap of size P_j to complete processing. This *j*-gap should be available before the arrival of its new job, otherwise τ_j will have a deadline miss and will be unschedulable. □

Lemma 5.3: *In the gap set $\sigma_j(\mathrm{T}|_t^{t+T_j})$ one element will be more than P_j for τ_j to be schedulable.*

Proof. The elements of $\sigma_j(\mathrm{T}|_t^{t+T_j})$ are all the *j*-gaps formed between two successive jobs of τ_j. From lemma 5.2 we know that one of these *j*-gaps, or the elements of $\sigma_j(\mathrm{T}|_t^{t+T_j})$ will be larger than P_j for τ_j to be schedulable. □

The mechanism of this method works as follows: Let, task set $\Gamma_n = \{\tau_1, \tau_2,\ldots,\tau_n\}$. We have to determine the response time of the first job of τ_j (RT_j) $(j < n)$. Without loss of generality we assume all tasks are released at the same time as τ_j (time 0). From lemma 5.1, we know that τ_j can only be processed inside the elements of the set

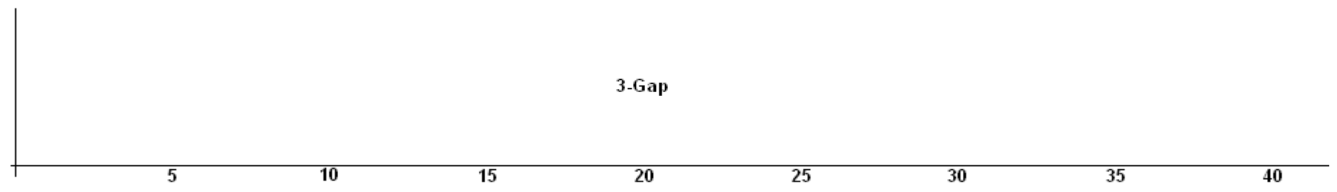

Figure 2(a): 3-gap available for processing of τ_3, $\sigma_3(T|_0^{40}) = \{[0,40)\}$

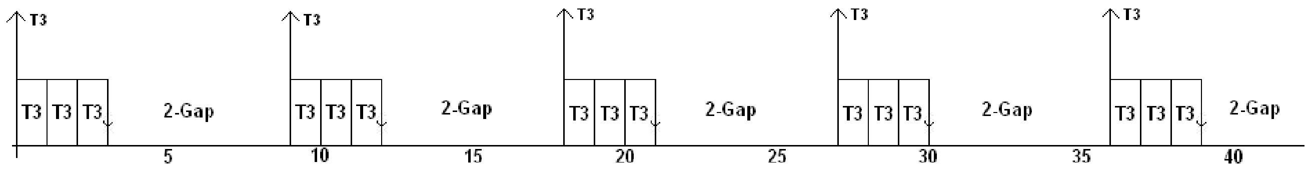

Figure 2(b): 2-gaps available for processing of τ_2, $\sigma_2(T|_0^{40}) = \{[3,9), [12,18), [21,27), [30,36), [39,40)\}$

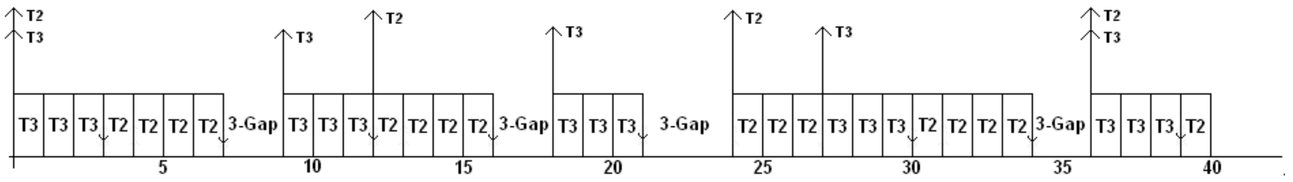

Figure 2(c): 1-gaps available for processing of τ_1, $\sigma_1(T|_0^{40}) = \{[7,9), [16,18), [21,24), [34,36)\}$

$\sigma_j(T|_0^{T_j})$. These elements are all the j-gaps available after the processing of tasks τ_n to τ_{j+1}. From lemma 5.2 we know that one of the j-gaps in the time interval $T|_0^{T_j}$, has to be larger than P_j for τ_j to be schedulable. We will first find the set $\sigma_j(T|_0^{T_j})$, and then search through this set, to find the first j-gap which is larger than P_j. τ_j will be processed in this j-gap making the response time of τ_j equal to $t_1 + P_j$, where t_1 is the threshold of this j-gap.

To find $\sigma_j(T|_0^{T_j})$ we progressively analyze gap sets of all higher priority tasks. The n-gap that is available for τ_n to run, is the entire length of the time interval $T|_0^{T_j}$. Hence, $\sigma_n(T|_0^{T_j}) = \{[0, T_j)\}$. The first job of τ_n will be released at time 0, and the second at time T_n. The mth job of τ_n will be released at $(m–1)\cdot T_n$. The $(n–1)$-gap left between the 1st and 2nd job is $[P_n, T_n)$. Similarly the $(n–1)$-gap left between the 2nd and 3rd job is $[T_n+P_n, 2\cdot T_n)$. Therefore, $\sigma_{n-1}(T|_0^{T_j}) = \{[P_n, T_n), [T_n+P_n, 2\cdot T_n)\ldots, [(m–2)\cdot T_n, (m–1)\cdot T_n)\}: (m–1)\cdot T_n \leq T_j$.

We see that the gap set $\sigma_{n-1}(T|_0^{T_j})$ is created after accounting for the processing of all jobs of τ_n, in the gap set $\sigma_n(T|_0^{T_j})$. Hence, the gap set $\sigma_n(T|_0^{T_j})$ has been *transformed* by the processing of all jobs of τ_n to result in $\sigma_{n-1}(T|_0^{T_j})$. We use the gap transformation function to account for the processing of the current task and get the gap set for the next lower priority task. Or,

$$\sigma_{n-1}(T|_0^{T_j}) = \lambda(\sigma_n(T|_0^{T_j}), \Gamma_n).$$

From lemma 5.1, we know that τ_{n-1} can only be processed in the gaps present in $\sigma_{n-1}(T|_0^{T_j})$. When we process all jobs of task $(n–1)$ in $T|_0^{T_j}$, some of the $(n–1)$ gaps present in $\sigma_{n-1}(T|_0^{T_j})$ will be used or reduce in size, leading to the formation of $(n–2)$-gaps. Hence, after accounting for the processing of all jobs of τ_{n-1} in $T|_0^{T_j}$, the gap set $\sigma_{n-2}(T|_0^{T_j})$ is created. The gap-transformation function can also be used to get the set $\sigma_{n-2}(T|_0^{T_j})$. Hence,

$$\sigma_{n-2}(T|_0^{T_j}) = \lambda(\sigma_{n-1}(T|_0^{T_j}), \Gamma_n)$$

Similarly,

$$\sigma_{n-3}(\mathrm{T}|_0^{T_j}) = \lambda\,(\sigma_{n-2}(\mathrm{T}|_0^{T_j}),\Gamma_n)$$
$$\ldots$$
$$\sigma_j(\mathrm{T}|_0^{T_j}) = \lambda\,(\sigma_{j+1}(\mathrm{T}|_0^{T_j}),\Gamma_n)$$

Once $\sigma_j(\mathrm{T}|_0^{T_j})$ is available we use the gap search function to give us the first j-gap in which τ_j can complete processing. Hence,

$$[t_1, t_2) = \mu(\sigma_j(\mathrm{T}), P_j).$$

Therefore,

$$RT_j = t_1+P_j$$

Let us illustrate this method by a simple case. Consider the example given in Section 3. Here, $\Gamma_3 = \{\tau_1, \tau_2, \tau_3\}$ and T_1,T_2,T_3 are 40,12,9 respectively. The processing times P_1,P_2,P_3 are 3,4,3 respectively and all tasks are released at time 0. We have to determine the actual response time for τ_1.

In the time interval $\mathrm{T}|_0^{40}$, the 3-gap available to process τ_3 is the entire length of the time interval period. Therefore, $\sigma_3(\mathrm{T}|_0^{40}) = \{[0,40)\}$ (*Figure 2(a)*). τ_3 will be processed at times 0,9,18,27 and 36 leaving 2-gaps in between each job. Therefore, $\sigma_2(\mathrm{T}|_0^{40}) = \{[3,9), [12,18), [21,27), [30,36), [39,40)\}$ (*Figure 2(b)*). The 1st, 2nd and 3rd jobs of τ_2 are processed in the 2-gaps [3,9), [12,18) and [21,27) respectively, while the 4th will start processing the 2-gap [39,40). Hence, $\sigma_1(\mathrm{T}|_0^{40}) = \{[7,9), [16,18), [21,24), [34,36)\}$ (*Figure 2(c)*). Since the length of the 1-gap [21,24) is more or equal to P_1, τ_1 will complete processing in this gap. Therefore,

$$RT_1 = 21 + 4 = 24.$$

VI. Algorithm to Determine Actual Response Time

We now present an algorithm that can determine the actual response time of τ_j, using the gap-enumeration method. The pseudo-code of the algorithm is given below. The algorithm takes Γ_n and τ_j as input and returns the actual response time of τ_j. In line 3, we assign an initial value to $\sigma_n(\mathrm{T}|_0^{T_j})$. Between lines 4 to 7, we successively compute the gap sets $\sigma_{n-1}(\mathrm{T}|_0^{T_j})$ to $\sigma_j(\mathrm{T}|_0^{T_j})$. Once the gap set for τ_j is known, we retrieve the earliest j-gap larger than P_j, using the gap search function $\mu(\mathrm{T}|_0^{T_j}, P_j)$ (line 8), and then compute the response time of τ_j (line 10).

If k-gaps to process lower priority tasks are not present, then the task set is not schedulable. In line 6, we check if gaps to process the lower priority task are present. If an i-gap to process a task τ_i is not present –1 is returned, signifying that the task set is not schedulable. A similar check in line 9 returns –1, if no j-gap is found to run τ_j.

Algorithm 5.1

```
1.  input: $\Gamma_n$, $\tau_j$
2.  output: $RT_j$ or -1
3.  $\sigma_n(\mathrm{T}|_0^{T_j}) \leftarrow \{[0,T_j)\}$
4.  loop $\tau_i \leftarrow n$ to $j$+1
5.      $\sigma_{i-1}(\mathrm{T}|_0^{T_j}) \leftarrow \lambda\,(\sigma_i(\mathrm{T}|_0^{T_j}),\Gamma_n)$
6.      if($|\sigma_{i-1}(\mathrm{T}|_0^{T_j})|$ = 0) return -1
7.  end loop
8.  $[t_1,t_2) \leftarrow \mu(\sigma_j(\mathrm{T}|_0^{T_j}), P_j)$
9.  if($t_1$ < 0) return -1
10. $RT_j = t_1 + P_j$
```

```
11. return $RT_j$
```

VI.I Gap-Enumeration with Dynamic Window Size

Algorithm 5.1 enumerates all the gaps present in the time window $\mathrm{T}|_0^{T_j}$. In certain cases, the time window $\mathrm{T}|_0^{T_j}$ could be large, and a much higher number of gaps than required could be enumerated. If $\mathrm{T}|_0^{T_j}$ is divided into smaller slices, the gap-enumeration algorithm can be made more efficient. We divide the time window $\mathrm{T}|_0^{T_j}$ into m windows ($1 \le m \le T_j$), of size $\left\lceil \frac{T_j}{m} \right\rceil$ and enumerate the gaps starting from window $\mathrm{T}|_0^{\left\lceil \frac{T_j}{m} \right\rceil}$. If no j-gap to run τ_j is found, then the length of the window can be progressively expanded by $\left\lceil \frac{T_j}{m} \right\rceil$. A modified form of algorithm 5.1, which uses dynamic size windows is given below. A new loop between lines 4 and 14 has been added, and the j-gap is searched in the time window $\mathrm{T}|_0^{L}$ where L has an initial value of $\left\lceil \frac{T_j}{m} \right\rceil$ (line 3). If a j-gap is found in $\mathrm{T}|_0^{L}$, the response time is returned (line 13), else L is incremented (line 13) and the j-gaps in the new time window are analyzed. If the j-gap is not found in the maximum possible time window $\mathrm{T}|_0^{L}$, –1 is returned signifying the unschedulability of the task τ_j.

Algorithm 5.2

```
1.  input: $\Gamma_n$, $\tau_j$, $m$
2.  output: $RT_j$ or -1
3.  $L \leftarrow \left\lceil \frac{T_j}{m} \right\rceil$
4.  loop while ($L < T_j + \left\lceil \frac{T_j}{m} \right\rceil$)
5.      $\sigma_n(\mathrm{T}|_0^{L}) \leftarrow \{[0,T_j)\}$
6.      loop $\tau_i \leftarrow n$ to $j+1$
7.          $\sigma_{i-1}(\mathrm{T}|_0^{L}) \leftarrow \lambda\ (\sigma_i(\mathrm{T}|_0^{L}),\Gamma_n)$
8.          if($|\sigma_{i-1}(\mathrm{T}|_0^{L})| = 0$) return -1
9.      end loop
10.     $[t_1,t_2) \leftarrow \mu(\sigma_j(\mathrm{T}|_0^{L}), P_j)$
11.     if($t_1 > 0$) $RT_j = t_1 + P_j$
12.     if $RT_j < T_j$ return $RT_j$
13.     $L \leftarrow L + \left\lceil \frac{T_j}{m} \right\rceil$
14. end loop
15. return -1
```

One obvious inefficiency of this approach is that, if a j-gap is not found in the time window $\mathrm{T}|_0^{L}$, the gaps have to be enumerated again in the time window of the next iteration $\mathrm{T}|_0^{2\cdot L}$. Developing ways to enumerate and search j-gaps only in the expanded section of the window ($\mathrm{T}|_L^{2\cdot L}$), rather than the whole window ($\mathrm{T}|_0^{2\cdot L}$) is a scope for future work.

VI.II Gap-Transformation Function

The gap transformation function $\lambda(\sigma_i(\mathrm{T}|_0^L),\Gamma_n)$, for a task τ_i, is an important component in determining the response time of tasks in P-FRP. It analyzes the gap-set $\sigma_i(\mathrm{T}|_0^L)$ for gaps in which τ_i could be processed, changes those gaps and returns the transformed gap-set. The pseudo-code for the implementation of this function is given below. The loop defined in lines 4-22 iterates for the number of jobs of τ_i in $\mathrm{T}|_0^L$. The loop inside lines 5-21 iterates for the number of gaps. We check for the *i*-gap in which job *q* of τ_i will process (line 8) and accordingly modify the gap. If the job *q* is after the threshold of this gap, then the *i*-gap will reduce in size (condition in lines 11 and 18) or split in two (condition in line 14). If the size of the *i*-gap is equal to or less than P_i, the *i*-gap will cease to exist since it will be consumed by the processing of τ_i. If job *q* completes processing in one of the gaps the inner loop exits (lines 13,17), and the search for *i*-gap to process the next job is started. If an *i*-gap to run job *q* is not found before the arrival of the next job, then τ_i is not schedulable and an empty gap set is returned by the function (line 6). The add operation will add the *k*-gap $[t_1, t_2)$ only if $t_1 \neq t_2$.

Algorithm 5.3

```
1.  input: $\sigma_i(\mathrm{T}|_0^L),\Gamma_n$
2.  output: $\sigma_{i-1}(\mathrm{T}|_0^L)$
3.  $jobs_i = \left\lceil \frac{L - \Phi_i}{T_i} \right\rceil + 1$
4.  loop job $q \leftarrow 1, jobs_i$
5.    loop k-gap $[t_1,t_2) \leftarrow 1, |\sigma_i(\mathrm{T}|_0^L)|$
6.      if $t_1 > t + T_i$ return $(|\sigma_{i-1}(\mathrm{T}|_0^L)| = 0)$

7.    if($t < t_1$) $t = t_1$
8.      if($t_1 \le t < t_2$)
9.      {
10.       remove$[t_1,t_2)$ from $\sigma_i(\mathrm{T}|_0^L)$
11.       if($t + P_i = t_2$)
12.        add $[t_1,t)$ to $\sigma_i(\mathrm{T}|_0^L)$
13.        exit loop gap
14.       if ($t + P_i < t_2$)
15.          add $[t_1,t)$ to $\sigma_i(\mathrm{T}|_0^L)$
16.        add $[t+P_j, t_2)$ to $\sigma_i(\mathrm{T}|_0^L)$
17.          exit loop gap
18.       if($t+P_j > t_2$)
19.          add $[t_1,t)$ to $\sigma_i(\mathrm{T}|_0^L)$
20.      }
21.   end loop
22. end loop
23. $\sigma_{i-1}(\mathrm{T}|_0^L) = \sigma_i(\mathrm{T}|_0^L)$
24. return $\sigma_{i-1}(\mathrm{T}|_0^L)$
```

VI.III Gap-Search Function

The gap search function $\mu(\sigma_k(\mathrm{T}|_0^L), P_k)$ does a simple search on $\sigma_k(\mathrm{T}|_0^L)$ and retrieves the first *k*-gap whose size is larger than P_k. The algorithm for the search depends on the type of data structure used to store the gaps.

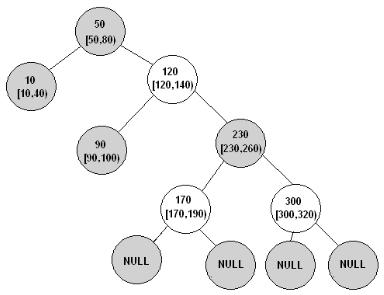

Figure 3: RB-tree for sample gap set. The shaded nodes denote a black node while the non-shaded are red nodes. The null nodes do not contain any data

---

Due to its guaranteed bounds for search and insertion time, we use a red-black tree (RB-tree) [6] to store gaps. A red-black tree, is a self balancing binary tree where each node has a color attribute of red or black. Other properties of a RB-tree are:

- The root node is black
- All leave nodes are black
- Children of every red node are black
- Path from leaf to root contain same number of black nodes

The gaps are stored in a RB-tree with threshold as the index. *Figure 3* shows the RB-tree for a sample gap set: $\sigma_k(\mathrm{T}|_0^{320})$={[10,40), [50,80), [90,100), [120,140), [170,190), [230,260), [300,320)}. The search function $\mu(\sigma_k(\mathrm{T}), P_k)$ is reduced to transversing the RB-tree from the left most leaf node (earliest gap), to the right most leaf node. The search order for the sample set based on node index is 10, 50, 90, 120, 170, 230, 260, 300.

VI.IV Time Complexity

We present an analysis of the time complexity of algorithm 5.2. The guaranteed worst case complexity for search, insertion and deletion operation in a RB-tree is O($\log_2 m$) (*log₂* is represented by *log* in the rest of the paper), where m is the number of elements of the tree. The transversal of the tree can take O(m) time, hence, the complexity of $\mu(\sigma_j(\mathrm{T}|_0^L), P_j)$ is bounded by $O(|\sigma_j(\mathrm{T}|_0^L)|)$.

Algorithm 5.2 uses algorithm 5.3 for gap-transformation. In algorithm 5.3 the loop in line 4 transverses for every gap, while the loop in line 3 transverses for every job of higher priority task τ_i in $\mathrm{T}|_0^L$. In the worst case, i-gap to run the job will be found at the end, hence, the total number of such transversals is $|\sigma_i(\mathrm{T}|_0^L)|\cdot(jobs_i + 1)$. When the i-gap to process the job is found, the existing i-gap is removed from $\sigma_i(\mathrm{T}|_0^L)$. Each removal operation is bounded by O(log m), where m is the number of gaps stored in the RB-tree. Then the modified i-gap is added back to the tree. One of the conditions, splits an available i-gap into two, in which case two additions will have to be done. Assuming that for every job two gaps are added, the worst case total time taken to add gaps for each job is:

$1 + \log(2) + \ldots. + \log(2\cdot m) = 1 + \log(2\cdot 3\cdot\ldots 2\cdot m)$

which is bounded by $O(\log(!2\cdot m))$. The total time for deletion and 2 additions is:

$$O(\log(!2\cdot m)) + O(\log m)$$

$O(\log(!2\cdot m))$ is the dominating term, hence, the total time for deletion and worst-case addition is: $O(\log(!2\cdot m))$.

Since, for the worst case every available i-gap is divided into 2, the number of gaps in $\sigma_{i\text{-}1}(\mathrm{T}|_0^L)$ will be double that of $\sigma_i(\mathrm{T}|_0^L)$. Hence, $m = 2\cdot|\sigma_i(\mathrm{T}|_0^L)|$. If the maximum number of additions is done for every i-gap and every job, then the complexity of the gap search function is bounded by:

$$O(|\sigma_i(\mathrm{T}|_0^L)|\cdot jobs_i \cdot\log(!2\cdot 2\cdot|\sigma_i(\mathrm{T}|_0^L)|))$$

In algorithm 5.2, the gap transformation function is called for τ_n to τ_{j+1}, and is executed for $n–j$ steps. Therefore, the total complexity for algorithm 5.2, is bounded by the time complexity for gap transformation and gap search functions and is given by the equation:

$$O((n\text{-}j)\cdot|\sigma_i(\mathrm{T}|_0^L)|\cdot jobs_i\cdot\log(!2\cdot 2\cdot|\sigma_i(\mathrm{T}|_0^L)|)) + O(|\sigma_j(\mathrm{T}|_0^L)|)$$

The gap transformation function is the dominant factor in the computation, therefore, the bound is:

$$O((n\text{-}j)\cdot|\sigma_i(\mathrm{T}|_0^L)|\cdot jobs_i\cdot\log(!2\cdot 2\cdot|\sigma_i(\mathrm{T}|_0^L)|))$$

The values of $jobs_i$ and $|\sigma_i(\mathrm{T}|_0^L)|$ is maximized when $T_i = \min(T_{n-j}, T_{n-j+1} \ldots T_n)$. Hence, τ_i represents the task with lowest arrival period among tasks having higher priority than j. The maximum possible value of L can be $T_j + \left\lceil \frac{T_j}{m} \right\rceil$, in which case the gaps will have to be enumerated m times, where each enumeration is bounded by:

$$O((n\text{-}j)\cdot|\sigma_i(\mathrm{T}|_0^L)|\cdot jobs_i\cdot\log(!2\cdot 2\cdot|\sigma_i(\mathrm{T}|_0^L)|))$$

The maximum possible value of m is T_j ($\left\lceil \frac{T_j}{m} \right\rceil$ =1). Accounting for each enumeration and putting the maximum value of L and m, the bound for the gap enumeration algorithm is:

$$O(T_j\cdot(n\text{-}j)\cdot|\sigma_i(\mathrm{T}|_0^{T_j+1})|\cdot jobs_i\cdot\log(!2\cdot 2\cdot|\sigma_i(\mathrm{T}|_0^{T_j+1})|))$$

It should be noted that the gap transformation function exits the loop (which iterates through the gaps), when the k-gap to run a job is found. For the upper bound, we have assumed that the loop continues till the last gap. Hence, the costs of additions and deletion are assumed much higher. The condition assumed for the worst-case will rarely be reached, hence, the general performance of the algorithm will always be better than what is defined by this bound.

Example: We ran the example given in section 3 through the time-accurate simulation algorithm, and computed the value of computational steps, which comes to 33.

VII. Analysis

Since the Time-accurate simulation (TAS) method is the only other known method for computing actual response time in P-FRP ,we present an experimental analysis of the performance of the Gap-enumeration (GE) algorithm, relative to TAS. For every addition and deletion operation into the RB-tree, the computational step is incremented by log(m), where m is the dynamically changing size of the RB-tree. Using computational steps for performance measurement is sufficient for this analysis, as it gives us a distinct idea of time that each algorithm will take to give the desired results.

We randomly generated 3 groups (group A, group B and group C) of 500 schedulable task sets. Group A has 3 tasks, group B had 5 and group C, 7 tasks. Each of the task sets in each group is unique in the sense, that at least 1 task is different between any two task sets in a group. The arrival period for each of the tasks in all the 3 groups were in the range [40,60), while the processing times were in the range [4,10). All tasks were assumed to be released simultaneously and the response time of the lowest priority task (τ_1) in each group was determined using the TAS and GE algorithms. In the GE algorithm, m was set to 1 for this analysis.

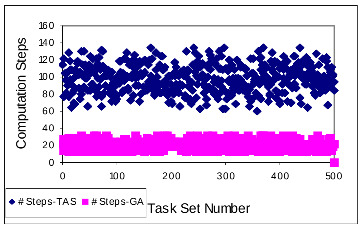

Figure 4(a): Computation steps for GA and TAS algorithm for task sets with 3 tasks

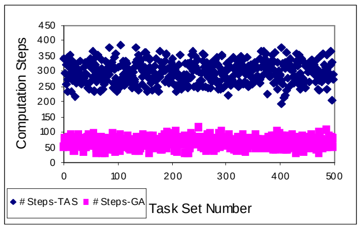

Figure 4(b): Computation steps for GA and TAS algorithm for task sets with 5 tasks

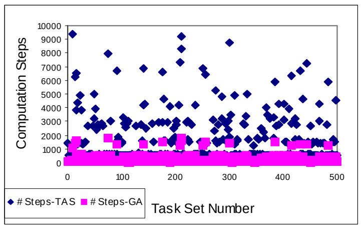

Figure 4(b): Computation steps for GA and TAS algorithm for task sets with 7 tasks

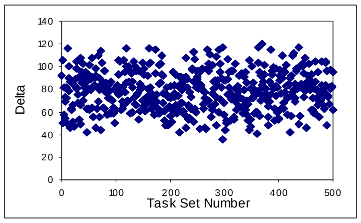

Figure 5(a): *Delta* (Steps TAS – Steps GA) for tasks sets with 3 tasks

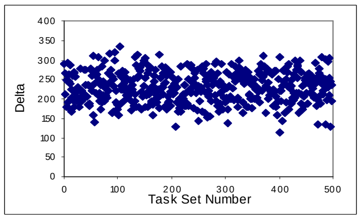

Figure 5(b): *Delta* (Steps TAS – Steps GA) for tasks sets with 5 tasks

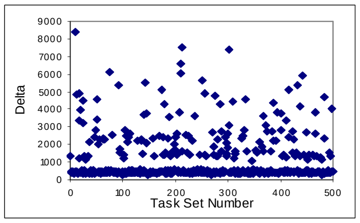

Figure 5(c): *Delta* (Steps TAS – Steps GA) for tasks sets with 7 tasks

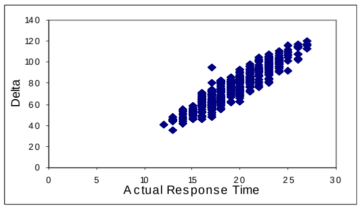

Figure 6(a): *Delta* (Steps TAS – Steps GA) vs. response time for tasks sets with 3 tasks

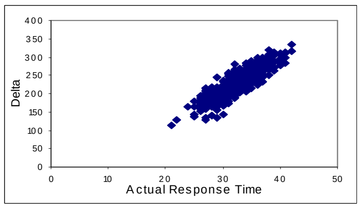

Figure 6(b): *Delta* (Steps TAS – Steps GA) vs. response time for tasks sets with 5 tasks

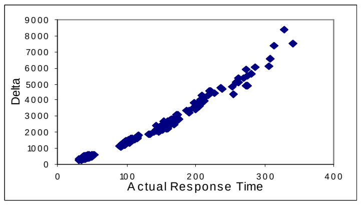

Figure 6(c): *Delta* (Steps TAS – Steps GA) vs. response time for tasks sets with 7 tasks

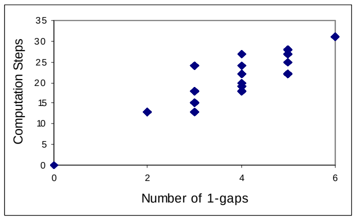

Figure 7(a): Number of total 1-gaps Vs. computational steps for GA method with 3 tasks

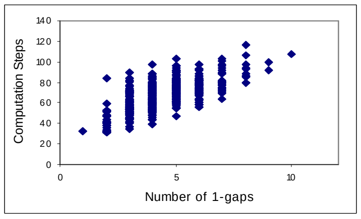

Figure 7(b): Number of total 1-gaps Vs. computational steps for GA method with 5 tasks

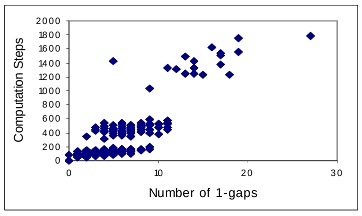

Figure 7(c): Number of total 1-gaps Vs. computational steps for GA method with 7 tasks

Figures 4(a),(b) and *(c)* shows the computational steps taken by the TAS and GE algorithms to compute the response time of task 1 in groups A, B and C respectively. It can be seen clearly that GE takes less number of computation steps as compared to the TAS algorithm. To get a precise idea on the difference between the computational steps taken by the TAS and GE algorithms, we show the difference in computational steps between the TAS and GE algorithms in *figures 5(a),(b)* and *(c)*. The Δ in the *y*-axis is given as:

Δ = Computational Steps in TAS - Computational steps in GE

The delta values tend to increase as the number of tasks present in the set increase. This could be attributed to a generally larger response time when the number of tasks are high. In *figures 6(a), (b)* and *(c)* we show the relation between response time and Δ. It is clear, that as the response time increases the GE algorithm becomes much more efficient relative to TAS.

In *Figure 7(a),(b) and (c)* we show the change in computational steps with number of 1-gaps present before the actual response time is found. The computational steps show a trend to increase with the number of total 1-gaps. A higher number of 1-gaps will increase the cost of search, insertion and deletion operations. However, for the same number of computational gaps, the computational time also varies. This is because even though the number of 1-gaps are the same, the response time for τ_1 in the task sets could vary, affecting the number of computation steps required.

VIII. Related Work

Response time analysis was first studied by Joseph and Pandya [15] and fixed priority scheduling was independently studied by Audsley et al [2]. In [2], an iterative method to compute actual response time of a preemptive system has been given. Kaibachev et al [16] present a basic response time analysis for P-FRP by placing restrictions on execution times of higher priority tasks. The authors have derived the response time bound of a task as equal to its arrival period. Ras and Cheng [21] have presented response time analysis , and have compared the performance of P-FRP execution with priority inversion strategies. The authors present a method to derive upper bound on response time by extending the iterative method developed by Audsley et al [2]. However, as shown in this paper, this method is unusable for most task sets. The flaw is that the authors make explicit assumptions on the abort pattern of higher priority tasks. The abort pattern is different for individual task sets and cannot be generally applied. Both [16], [21] do not define any method to compute actual response times for P-FRP.

Transactional memory systems have been described by Herlihy and Moss [14]. Response time analysis for transaction memory using dynamic scheduling for multiprocessor systems has been done by Fahmy et al [9]. In Manson et al [18] an atomic processing of the critical section has been implemented in Java. In Manson et al's paper response time analysis using fixed priority scheduling has been done, however the response time so derived does not account for complete re-processing of critical sections. Davis and Burns [7] derive upper bounds on response time for fixed priority scheduling building upon the work done by Bini and Baruah [3]. Anderson et al [1] do response time analysis of the lock-free mechanism. Lock-free is a mechanism to avoid priority inversion [23] the implementation of which is via an unconditional loop that terminates when the necessary updates to the shared resource are complete. The schedulability conditions given for fixed-priority scheduling in [1] assume a constant 'extra computation time' in case of a failed update. If we consider this equivalent to an abort cost in P-FRP it cannot be a constant as the abort cost varies for every task. Comparisons between transaction memory based systems and lock-free processing and benefits of the former have been shown in Herlihy and Moss [14].

Chen et al [5] have investigated the presence of feasible intervals in which a job can process. A task can start processing and if the interval is less than its processing time then it will restart at the next available interval. This model is similar to P-FRP if intervals are considered as *k*-gaps. The authors show that scheduling such a task set is an NP-hard problem, and give approximation algorithms for scheduling the tasks. Algorithms for preemptive and non-preemptive task sets are given, where the non-preemptive tasks have to complete processing in one time interval. Byun et al [4] adopt the critical section approach for CPU tasks, into a Database model. A high priority transaction can abort a low priority transaction if they share a lock and the low priority task is restarted. The authors have bounded the response time for a task and have considered the cost of re-execution.

IX. Conclusions and Future Work

A common method for determining actual response time in the preemptive model cannot be applied to the execution model of P-FRP, due to the abort of preempted tasks. A straightforward method way to compute actual response time, is to run a time accurate simulation of the P-FRP execution model. The time complexity for this method is defined by the length of time for which the simulation has to be run. In several practical situations, the length of time to analyze the system could be very large. Also during the design phase of a real-time system, as well as for determining the worst-case release time, many scenarios need to be analyzed for their response time. Using a time-accurate simulation will take significant amount of computation time, making its use impractical in these situations.

The gap-enumeration method is a different approach for computing actual response time in the P-FRP execution model. Comparisons with the time-accurate method show that the gap-enumeration method is much more efficient than the former. For P-FRP systems with numerically higher response times, the gap-enumeration method offers engineers a fast alternative for the computation of actual response times. The performance of this method is directly proportional to the number of *k*-gaps present in the system. The number of *k*-gaps has no impact on the time accurate simulation method, whose computational complexity is primarily governed by the time steps that have to be covered. While the gap-enumeration algorithm is faster than the time-accurate simulation, it is clearly not as efficient as Audsley's method. However, we feel that due to the abort nature of tasks, computing response time using fixed iterations on a mathematical expression, as developed by Audsley et al, might not be feasible for P-FRP. Hence, algorithm based approaches, such as the gap-enumeration method, are perhaps, the only way to compute actual response time in P-FRP.

We have presented the gap-enumeration algorithm in its simple form. Several changes could be made to improve the efficiency of this method. The main computational cost incurred by the gap-enumeration method is during insertion, deletion and search of the data structure used to store k-gaps. A hash table could be used in conjunction with the RB-tree to index the locations of k-gaps thereby making the search, insertion and deletion operation more efficient. In ongoing work, we are also exploring a method where 2-dimensional array is used to keep track of gaps created for each task. In this work the values of $t_{copy}(k)$ and $t_{restore}(k)$ for a task τ_k, have been considered as 1. This makes our execution model simple since we do not have to account for situations when a lower priority task cannot be preempted. This work will be enhanced by considering more probable values of $t_{copy}(k)$ and $t_{restore}(k)$. Computing actual response time in P-FRP, when task arrivals are sporadic is also an open area of research. This gap-enumeration algorithm can also be modified to determine actual response time in the preemptive model. This can be achieved by changing the gap-transformation and gap-search functions such that, rather than waiting for a k-gap larger than or equal to P_k being available, τ_k can complete processing as soon as the combined length of all gaps (starting from the first gap) equals P_k. Of course, the usefulness and relative performance of this modified gap-enumeration algorithm vis-à-vis Audsley's method will have to be ascertained.

While the classical preemptive model of execution is well understood and several mature studies have been done over several years, the abort-restart model of execution has not been deeply investigated. Therefore, this study is valuable in response time studies for systems having similar execution models such as in real-time databases [4], transactional memory systems [14], lock-free execution [1] and Java [18].

References

[1] J. H. Anderson, S. Ramamurthy, K. Jeffay. "Real-time computing with Lock-free Shared Objects". *ACM Transactions on Comp.Sys. 5(6), pp.388-395*, 1997

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Wellings. "Applying new scheduling theory to static priority preemptive scheduling". *Software Engineering Journal 8(5), pp: 284-292*, 1993

[3] E. Bini, S.K. Baruah. "Efficient Computation of Response Time Bounds under Fixed-priority Scheduling". *Proc. Of the 15th conference on Real-Time and Network Systems, pp. 95-104*,2007

[4] J. Byun, A. Burns, A. Wellings. "A Worst-Case Behavior Analysis for Hard Real-time transactions". *Workshop on Real-time Databases*, 1996

[5] J. J. Chen, J. Wu, C. S. Shih, T.W. Kuo. "Approximation algorithms for Scheduling Multiple Feasible Interval Jobs". *RTCSA'05 , pp.: 11 - 16* ,2005

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, Chapter 13: Red-Black Trees, pp.273–301*, 2001

[7] R.I. Davis, A.Burns. "Response Time Upper Bounds for Fixed Priority Real-Time Systems".*RTSS'08, pp.407-418*, 2008

[8] C. Elliott, P. Hudak. "Functional reactive animation". *ICFP'97,pp/ 263-273*, 1997

[9] Erlang, http://www.erlang.org

[10] S.F. Fahmy, B. Ravindran, E.D. Jensen. "Response time analysis of software transactional memory-based distributed real-time systems", *ACM SAC Operating Systems*, 2009

[11] F#, http://research.microsoft.com/en-us/um/cambridge/projects/fsharp

[12] Haskell, *http://www.haskell.org*

[13] T. Hawkins. "Controlling Hybrid Vehicles with Haskell*", Commerical Uses of Functional Languages (CUSP)'08, 2008*

[14] M. Herlihy, J.E.B. Moss. "Transactional memory: architectural support for lock-free data structures". *ACM SIGARCH Computer Architecture New (Col. 21, Issue 2),pp. 289-300,* 1993

[15] M. Joseph, P. Pandya. "Finding Response Times in a Real-Time System". *BCS Computer Journal (Vol. 29, No. 5), pp: 390-395,* 1986

[16] R. Kaiabachev, W. Taha, A. Zhu. E-FRP with Priorities. *EMSOFT'07 , pp: 221-230* , 2007

[17] C. L. Liu, L. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". *Journal of the ACM (Volume 20 Issue 1), pp: 46 - 61* , 1973

[18] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, J. Vitek. "Preemptible Atomic Regions for Real-Time Java". *RTSS'05, pp.62-71,* 2005

[19] J. Peterson, G. D. Hager and P. Hudak. "A Language for Declarative Robotic Programming". *ICRA'99, IEEE*, 1999

[20] J. Peterson, P.Hudak, A.Reid, G. D. Hager. "FVision: A Declarative Language for Visual Tracking". In *Symposium on Practical Aspects of Declarative Languages*, 2001

[21] J. Ras, A. Cheng. "Response Time Analysis for the Abort-and-Restart Task Handlers of the Priority-Based Functional Reactive Programming (P-FRP) Paradigm". *RTCSA'09*, 2009

[22] M. F. Ringenburg, D. Grossman. "AtomCaml: first-class atomicity via rollback". *ACM ICFP'05, pp. 92-104,* 2005

[23] L. Sha, R. Rajkumar, J. P. Lehoczky. "Priority Inheritance Protocols: An approach to Real Time Synchronization". *Transactions on Computers Volume 39, Issue 9, pp:1175 – 1185,* 1990

[24] Z. Wan, W. Taha, and P. Hudak. "Real - time FRP". *ICFP'01, pp: 146-156, ACM Press* ,2001

[25] Z. Wan, W. Taha, and P. Hudak. "Task driven FRP". *PADL'02*, 2002

[26] Z. Wan and P. Hudak. "Functional reactive programming from first principles". *ACM SIGPLAN Conference on Programming Language Design and Implementation,pp.242-252,*2000