Determining Actual Response Time in P-FRP*

Chaitanya Belwal and Albert M.K. Cheng

Department of Computer Science,
University of Houston, TX, USA
{cbelwal,cheng}@cs.uh.edu

Abstract. A purely functional model of computation, called Priority-based Functional Reactive Programming (P-FRP), has been introduced as a new paradigm for building real-time software. Unlike the classical preemptive model[1] of real-time systems, preempted events in P-FRP are aborted and have to restart when higher priority events have completed, making the response time of events dependent on the execution pattern of higher priority events. Though methods to determine approximate values for the response time of P-FRP events have been presented, no convenient method has yet been established to determine actual response time. A common method for computing actual response time in the preemptive model does not give guaranteed results in P-FRP. A simulation based 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, and can be easily adopted to determine actual response time in P-FRP.

Keywords: real-time systems, embedded systems, response time, schedulability analysis, functional programming.

1 Introduction

Functional Reactive Programming (FRP) [22] is a declarative programming language for modeling and implementing reactive systems. It has been used for a wide range of applications, notably, graphics [7], robotics [15], and vision [16]. 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 [9], 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 [13,20,21]. Recently, it was shown that a variant called priority-based FRP (P-FRP)

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

1 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.

[13], 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, a requirement [19] in the functional programming model is that the state of the system cannot be changed, and no function can have side effects. Hence, to maintain this guarantee of 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 responsiveness of each handler is achieved.

This transactional execution model used in P-FRP is not new, and such models have been presented in the past. These are the transactional memory systems [11] and lock-free execution for critical sections [1]. 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 [18]. Studies on the temporal properties of the transactional model are being done by some research groups. However, the response time studies currently available [1,8] provide only basic schedulability analysis by modifying existing methods developed for the preemptive model. A study to find actual response time for this execution model has not been presented yet.

Previous work on P-FRP [13,17] provided basic results on schedulability and approximate upper bounds on response times. Though approximate upper bounds provide only a general idea on the schedulability of events, the methods to compute them are much faster [4,6,17]. In this paper we use the term 'actual' to differentiate from approximate or bounded response time. Actual response time is a more accurate indicator of the temporal properties of events in the system. Hence, actual response time is more useful when an accurate modeling 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 event in P-FRP 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 3 for example), and new methods for determining actual response time in P-FRP are required.

1.1 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 feasible to work out these response times by hand or ad-hoc 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.

After reviewing basic concepts and the P-FRP execution model (Section 2) we:

- Present Audsley's iterative method for computing actual response time in the preemptive model (Section 3)
- Present an enumeration technique for idle periods, which has been termed as the *gap-enumeration* method (Section 4)
- Present an algorithm that determines the actual response time of a task using the gap-enumeration method (Section 5)
- Provide performance analysis between the time accurate and gap-enumeration algorithms (Section 6)

And conclude by reviewing related work (Section 7) and a reflection on these results (Section 8).

2 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.

2.1 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
- 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 **restore** 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 m^{th} 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 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 **threshold** of the k-gap $[t_1, t_2)$ is time t_1
- $\mathrm{T}|_{t_1}^{t_2}$ represent the **time window** for analyzing gaps, such that: $\forall t\in \mathrm{T}|_{t_1}^{t_2}$, $t_1 \le t < t_2 \wedge t_1 \ne t_2$. 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** $\sigma_k(\mathrm{T}|_{t_1}^{t_2})$ contains all the unique k-gaps present in the time interval $\mathrm{T}|_{t_1}^{t_2}$. The k-gaps present in $\sigma_k(\mathrm{T}|_{t_1}^{t_2})$ are also disjoint:

  for any two gaps $[t_{x1},t_{y1}), [t_{x2},t_{y2}) \in \sigma_k(\mathrm{T}|_{t_1}^{t_2})$, if $t\in[t_{x1},t_{y1})$ then $t\notin [t_{x2},t_{y2})$
- $|\sigma_k(\mathrm{T}|_{t_1}^{t_2})|$ represents the number of k-gaps present in $\sigma_k(\mathrm{T}|_{t_1}^{t_2})$
- The **gap-transformation function** $\lambda(\sigma_k(\mathrm{T}|_{t_1}^{t_2}), \Gamma_n)$ 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:

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

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

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

  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 $\sigma_k(\mathrm{T}|_{t_1}^{t_2})$
- The **computational steps** of an algorithm is a numerical measurement of the number of times major iterations of the algorithm have been performed during execution. This value gives us a general idea of the performance of the algorithms considered in this paper
- The **response time** of a τ_k written as RT_k is the relative time after its release at 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.

2.2 Execution Model and Assumptions

In this study all tasks are assumed to execute in a uniprocessor system with no precedence constraints. When a job of 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 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.

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

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

2.2.2 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) [14] of the task being analyzed. In their seminal work, Liu and Layland [14] 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 all higher priority tasks. Or, the release offset of task τ_j and higher priority tasks is the same. This is also termed as a *synchronous* release of tasks. As shown in [3], for P-FRP, a *synchronous* release of τ_j and higher priority tasks is not guaranteed to result in the WCRT of τ_j.

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 known *a priori*. These release offsets, 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, have to be used to compute the actual release time under each of the possible release offset combinations. Finally, the highest value of the response time computed under each release offset combination will be the WCRT for the task.

3 Computing Actual Response Time in the Preemptive Model

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

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

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:

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

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 a task. As shown in [2], equation 3.1 can also be modified to determine response time when tasks have non-zero offsets (tasks are released *asynchronously*) or encounter *release jitter*.

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

Task	P	T
τ_1	20	100
τ_2	20	70
τ_3	10	40

We have to compute the response time of τ_1 using equation 3.1, assuming a synchronous release of tasks. The iterations of the computation are given below:

#1, n=0: $RT_1^1 = 20 + (\left\lceil \frac{0}{40} \right\rceil \cdot 10 + \left\lceil \frac{0}{70} \right\rceil \cdot 20) = 20$ #3, n=2: $RT_1^3 = 20 + (\left\lceil \frac{50}{40} \right\rceil \cdot 10 + \left\lceil \frac{50}{70} \right\rceil \cdot 20) = 60$

#2, n=1: $RT_1^2 = 20 + (\left\lceil \frac{20}{40} \right\rceil \cdot 10 + \left\lceil \frac{20}{70} \right\rceil \cdot 20) = 50$ #4, n=3: $RT_1^4 = 20 + (\left\lceil \frac{60}{40} \right\rceil \cdot 10 + \left\lceil \frac{60}{70} \right\rceil \cdot 20) = 60$

Since, $RT_1^3 = RT_1^4$, the iteration will terminate giving us the response time for τ_1 as 60. In *Figure 1* we show how P-FRP processes the tasks in the time window $T|_0^{100}$, resulting in the response time of τ_1 as 70. *Figure 1* also illustrates the fact that, even though the processing time of τ_1 if 20 and is known *a priori*, it takes a total processor time of 30 to complete processing due to an abort at time 40.

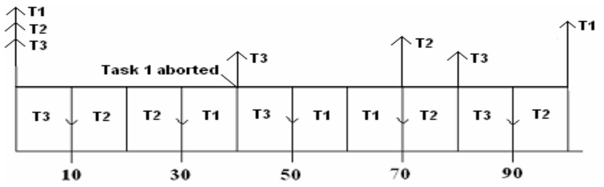

Fig. 1. Task execution graph showing τ_1 completing processing at time 70. T1, T2 and T3 represent tasks τ_1, τ_2 and τ_3 respectively.

3.1.1 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 [17]. An abort cost to the original equation has been added. The modified equation is given as:

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

hp_i represents the set of tasks having a higher priority than τ_i. The initial value for $WCRT_i$ is set to P_i. This equation computes 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 in P-FRP a synchronous release is not guaranteed to lead to WCRT. Applying equation 3.3 to our example, and setting $WCRT_1^0$ =20:

$$1:\ WCRT_1^1 = 20 + \left(\left\lceil \frac{20}{40} \right\rceil \cdot 10 + \left\lceil \frac{20}{70} \right\rceil \cdot 20\right) + \left\lceil \frac{20}{40} \right\rceil \cdot 20 + \left\lceil \frac{20}{70} \right\rceil \cdot 20 = 90$$

$$2:\ WCRT_1^2 = 20 + \left(\left\lceil \frac{90}{40} \right\rceil \cdot 10 + \left\lceil \frac{90}{70} \right\rceil \cdot 20\right) + \left\lceil \frac{90}{40} \right\rceil \cdot 20 + \left\lceil \frac{90}{70} \right\rceil \cdot 20 = 190$$

$$3:\ WCRT_1^3 = 20 + \left(\left\lceil \frac{190}{40} \right\rceil \cdot 10 + \left\lceil \frac{190}{70} \right\rceil \cdot 20\right) + \left\lceil \frac{190}{40} \right\rceil \cdot 20 + \left\lceil \frac{190}{70} \right\rceil \cdot 20 = 290$$

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.

A straightforward way for 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. Due to limited space, the pseudo-code for such an algorithm is given in [3]. The computational complexity of this algorithm is bounded by $O((T_j - P_j) \cdot (n-j)^2 \cdot T_k^2)$, derivation of which is also provided in [3].

4 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. Due to limited space, we have not given proofs and detailed pseudo-code of some of our methods. These are available in [3].

Lemma 4.1 [3, 5.1]. *A task τ_j can be processed only in elements of the set $\sigma_j(\mathrm{T}|_{t_1}^{t_2})$.*

Lemma 4.2 [3, 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.*

Lemma 4.3 [3, 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.*

The mechanism of the gap-enumeration 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, assume all tasks are released at the same time as τ_j (time 0). From lemma 4.1, we know that τ_j can only be processed inside the elements of the set $\sigma_j(\mathrm{T}|_0^{T_j})$. These elements are all the *j*-gaps available after the processing of tasks τ_n to τ_{j+1}. From lemma 4.2, we know that one of the *j*-gaps in the time interval $\mathrm{T}|_0^{T_j}$, has to be larger than P_j for τ_j to be schedulable. We will first find the set $\sigma_j(\mathrm{T}|_0^{T_j})$, and then search through this set for 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 $t_l + P_j$, where t_l is the threshold of this *j*-gap.

To find $\sigma_j(\mathrm{T}|_0^{T_j})$ 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 $\mathrm{T}|_0^{T_j}$. Hence, $\sigma_n(\mathrm{T}|_0^{T_j}) = \{[0, T_j)\}$. The first job of τ_n will be released at time 0, and the second at time T_n. The m^{th} 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, $\sigma_{n-1}(\mathrm{T}|_0^{T_j}) = \{[P_n, T_n), [T_n+P_n, 2\cdot T_n)\ldots, [(m-2)\cdot T_n, (m-1)\cdot T_n)\}: (m-1)\cdot T_n \le T_j$.

We see that the gap set $\sigma_{n-1}(\mathrm{T}|_0^{T_j})$ is created after accounting for the processing of all jobs of τ_n, in the gap set $\sigma_n(\mathrm{T}|_0^{T_j})$. Hence, the gap set $\sigma_n(\mathrm{T}|_0^{T_j})$ has been transformed by the processing of all jobs of τ_n to result in $\sigma_{n-1}(\mathrm{T}|_0^{T_j})$. 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,

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

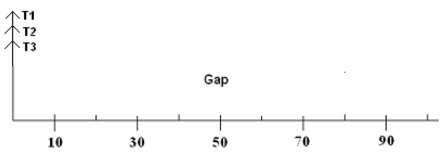

Fig. 2(a). 3-gap available for processing of τ_3, $\sigma_3(T|_0^{100}) = \{[0,100)\}$

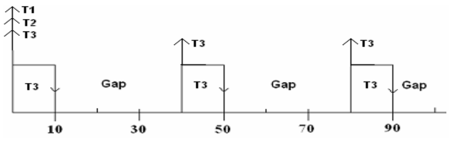

Fig. 2(b). 2-gaps available for processing of τ_2, $\sigma_2(T|_0^{100}) = \{[10,40), [50,80), [90,100)\}$

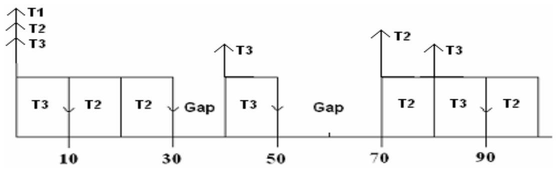

Fig. 2(c). 1-gaps available for processing of τ_1, $\sigma_1(T|_0^{100}) = \{[30,40), [50,70)\}$

From lemma 4.1, 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 τ_{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,

$$\sigma_{n-3}(T|_0^{T_j}) = \lambda\,(\sigma_{n-2}(T|_0^{T_j}),\Gamma_n)$$

$$\ldots$$

$$\sigma_j(T|_0^{T_j}) = \lambda\,(\sigma_{j+1}(T|_0^{T_j}),\Gamma_n)$$

Once $\sigma_j(T|_0^{T_j})$ 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 100,70,40 respectively. The processing times P_1,P_2,P_3 are 20,20,10 respectively and all tasks are released at time 0. We have to determine the actual response time for τ_1.

In the time interval $T|_0^{100}$, the 3-gap available to process τ_3 is the entire length of the time interval period. Therefore, $\sigma_3(T|_0^{100}) = \{[0,100)\}$ (*Figure 2(a)*). τ_3 will be processed at times 0,40 and 80 leaving 2-gaps in between each job. Therefore, $\sigma_2(T|_0^{100}) = \{[10,40), [50,80), [90,100)\}$ (*Figure 2(b)*). The first job of τ_2 is processed in the 2-gap [10,30), and the second job starts processing at time 70, but is aborted by second job of τ_3 at time 80. τ_2 will restart processing in the 2-gap [90,100). Hence, $\sigma_1(T|_0^{100}) = \{[30,40), [50,70)\}$ (*Figure 2(c)*). Since the length of the 1-gap [50,70) is more or equal to P_1, τ_1 will complete processing in this gap. Therefore,

$$RT_1 = 50 + 20 = 70.$$

5 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 $\sigma_n(T|_0^{T_j})$. Between lines 4 to 7, we successively compute the gap sets $\sigma_{n-1}(T|_0^{T_j})$ to $\sigma_j(T|_0^{T_j})$. Once the gap set for τ_j is known, we retrieve the earliest j-gap larger than P_j, using the gap search function $\mu(T|_0^{T_j}, P_j)$ (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.  $\sigma_n(T|_0^{T_j}) \leftarrow \{[0, T_j)\}$
4.  loop $i \leftarrow n$ to $j+1$
5.          $\sigma_{i-1}(T|_0^{T_j}) \leftarrow \lambda\ (\sigma_i(T|_0^{T_j}), \Gamma_n)$
6.          if($|\sigma_{i-1}(T|_0^{T_j})|$ = 0) return -1
7.  end loop
8.  $[t_1, t_2) \leftarrow \mu(\sigma_j(T|_0^{T_j}), P_j)$
9.  if($t_1$ < 0) return -1
10. $RT_j = t_1 + P_j$
11. return $RT_j$
```

5.1 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 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 can 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 is progressively incremented by $\left\lceil \frac{T_j}{m} \right\rceil$. A modified form of algorithm 5.1, which uses dynamic size windows is given in [3]. The time complexity of this algorithm is bounded by $O(T_j \cdot (n-j) \cdot |\sigma_i(\mathrm{T}\,|_0^{T_j+1})| \cdot jobs_i \cdot \log(!2 \cdot 2 \cdot |\sigma_i(\mathrm{T}\,|_0^{T_j+1})|))$, derivation of which is available in [3].

5.2 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 available in [3].

5.3 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. Due to its guaranteed bounds for search and insertion time, we use a red-black tree (RB-tree) [5] to store the gap. 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: $\sigma_k(\mathrm{T})=\{[10,40), [50,80), [90,100), [120,140), [170,190), [230,260), [300,320)\}$. 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, 300.

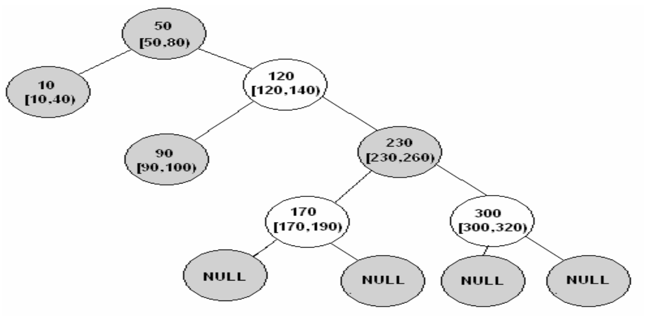

Fig. 3. RB-tree for sample gap set. The shaded nodes denote a black node while the non-shaded are red nodes. The null nodes do not contain any data.

6 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 in 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 (groups A, B and C) of 500 schedulable task sets. Task sets in group A have 3 tasks, group B, 5 tasks 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. The arrival period for each of the tasks in all the 3 groups were selected from the range [40,60], while the processing times were selected from [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.

The difference in computational steps between the TAS and GE algorithms for task set of sized 3,5 and 7 are shown in *figures 4(a),5(a) and 6(a)*. The Δ in the y-axis is given as:

$$\Delta = \text{Computational Steps in TAS - Computational steps in GE}$$

It can be seen clearly that GE takes less number of computation steps as compared to the TAS algorithm. 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 4(b), 5(b) and 6(b)* we show the relation between response time and Δ. It is clear, that as the response time increases, the delta values increase showing that the GE algorithm becomes much more efficient relative to TAS. Additional results of out analysis are available in [3].

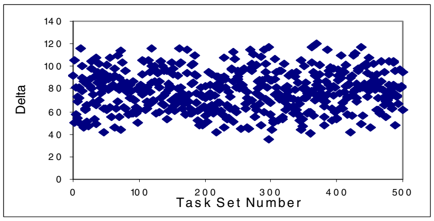

Fig. 4(a). *Delta* (Steps TAS – Steps GE) for tasks sets with 3 tasks

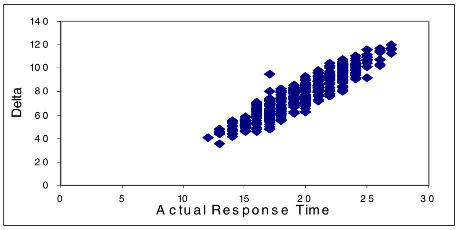

Fig. 4(b). *Delta* (Steps TAS – Steps GE) vs. response time for tasks sets with 3 tasks

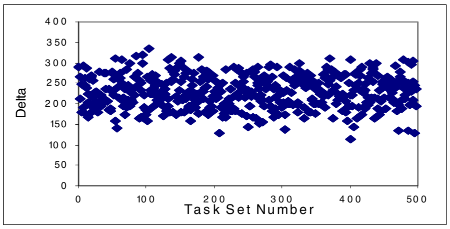

Fig. 5(a). *Delta* (Steps TAS – Steps GE) for tasks sets with 5 tasks

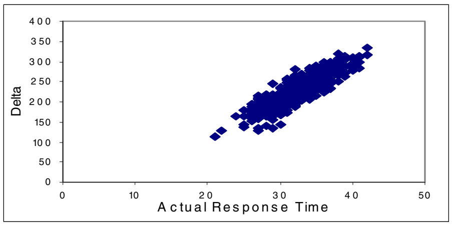

Fig. 5(b). *Delta* (Steps TAS – Steps GE) vs. response time for tasks sets with 5 tasks

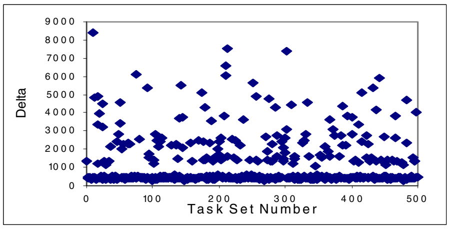

Fig. 6(a). *Delta* (Steps TAS – Steps GE) for tasks sets with 7 tasks

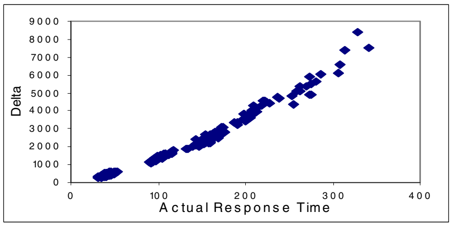

Fig. 6(b). *Delta* (Steps TAS – Steps GE) vs. response time for tasks sets with 7 tasks

7 Related Work

Response time analysis was first studied by Joseph and Pandya [12] and fixed priority scheduling was independently studied by Audsley et al [2]. In [2], an iterative method to compute actual response time in the preemptive model is given. Kaibachev et al [13] 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 [17] 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 cost from higher priority tasks. The abort cost is different for individual task sets and cannot be generally applied. Both [13, 17] do not define any method to compute actual response times for P-FRP.

Transactional memory systems have been described by Herlihy and Moss [11]. Response time analysis for transaction memory using dynamic scheduling for multiprocessor systems has been done by Fahmy et al [8]. Davis and Burns [6] derive upper bounds on response time for fixed priority scheduling building upon the work done by Bini and Baruah [4]. Anderson et al [1] do response time analysis of the lock-free mechanism. Lock-free is a mechanism to avoid priority inversion [18] 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 [11].

8 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 approach is to run a time accurate simulation of the P-FRP execution model. However the time complexity of this approach is high and, therefore it is not feasible in most practical 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 number of 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 [2]. 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 a 2-dimensional array is used to keep track of gaps created for each task.

References

1. Anderson, J.H., Ramamurthy, S., Jeffay, K.: Real-time computing with Lock-free Shared Objects. ACM Transactions on Comp. Sys. 5(6), 388–395 (1997)
2. Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.: Applying new scheduling theory to static priority preemptive scheduling. Software Engineering Journal 8(5), 284–292 (1993)
3. Belwal, C., Cheng, A.M.K.: Determining Actual Response Time in P-FRP. Technical Report: UH-CS-10-05, Dept. Of Computer Science, University of Houston (2010)
4. Bini, E., Baruah, S.K.: Efficient Computation of Response Time Bounds under Fixed-priority Scheduling. In: Proc. of the 15th Conference on Real-Time and Network Systems, pp. 95–104 (2007)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Red-Black Trees. In: Introduction to Algorithms, 2nd edn., ch. 13, pp. 273–301. MIT Press/McGraw-Hill (2001)
6. Davis, R.I., Burns, A.: Response Time Upper Bounds for Fixed Priority Real-Time Systems. In: RTSS 2008, pp. 407–418 (2008)
7. Elliott, C., Hudak, P.: Functional reactive animation. In: ICFP 1997, pp. 263–273 (1997)
8. Fahmy, S.F., Ravindran, B., Jensen, E.D.: Response time analysis of software transactional memory-based distributed real-time systems. ACM SAC Operating Systems (2009)
9. Hammond, K.: Chapter 1 – Is it Time for Real-Time Functional Programming. In: Gilmore, S. (ed.) Trends in Functional Programming, vol. 4. Intellect Ltd. (2005)
10. Haskell, `http://www.haskell.org`
11. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. ACM SIGARCH Computer Architecture New 21(2), 289–300 (1993)
12. Joseph, M., Pandya, P.: Finding Response Times in a Real-Time System. BCS Computer Journal 29(5), 390–395 (1986)
13. Kaiabachev, R., Taha, W., Zhu, A.: E-FRP with Priorities. In: EMSOFT 2007, pp. 221–230 (2007)
14. Liu, C.L., Layland, L.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM 20(1), 46–61 (1973)
15. Peterson, J., Hager, G.D., Hudak, P.: A Language for Declarative Robotic Programming. In: ICRA 1999. IEEE, Los Alamitos (1999)
16. Peterson, J., Hudak, P., Reid, A., Hager, G.D.: FVision: A Declarative Language for Visual Tracking. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, p. 304. Springer, Heidelberg (2001)
17. Ras, J., Cheng, A.: Response Time Analysis for the Abort-and-Restart Task Handlers of the Priority-Based Functional Reactive Programming (P-FRP) Paradigm. In: RTCSA 2009 (2009)
18. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority Inheritance Protocols: An approach to Real Time Synchronization. Transactions on Computers 39(9), 1175–1185 (1990)
19. Swaine, M.: It's Time to Get Good at Functional Programming. Dr. Dobbs Journal (December 2008), `http://www.drdobbs.com`
20. Wan, Z., Taha, W., Hudak, P.: Real - time FRP. In: ICFP 2001, pp. 146–156. ACM Press, New York (2001)
21. Wan, Z., Taha, W., Hudak, P.: Task Driven FRP. In: Adsul, B., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, p. 155. Springer, Heidelberg (2002)
22. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 242–252 (2000)