Partitioned Scheduling of P-FRP in Symmetric Homogenous Multiprocessors*

Chaitanya Belwal, Albert M.K. Cheng

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

UH-CS-11-01
Feburary 1, 2011

Keywords: Multi-processor scheduling, Response Time Analysis, Real-time Systems, Functional Programming

Abstract

Functional Reactive Programming (FRP), is a declarative approach to modeling and building reactive systems. Priority-based FRP (P-FRP) has recently been introduced as a FRP formalism that guarantees real-time response. P-FRP guarantees that when a higher priority task is released, the system will immediately preempt any executing lower-priority tasks. To maintain guarantees of state-less execution offered by a purely functional model of programming, P-FRP implements a transactional nature of execution. Each higher priority task in P-FRP can abort a lower priority task forcing it to restart. Existing work on partitioning tasks in multi-processor systems have been focused on the classical preemptive model† of execution. However, due do its transactional nature of execution, the schedulability tests used in the partitioning algorithms for the classical preemptive model, cannot be applied 'as is' to the P-FRP execution model. While multiprocessor response time analysis of P-FRP has been done in previous work, partitioning schemes for tasks in multi-processor systems have not been presented yet. In this paper, we present an exact schedulability test for P-FRP and use it in two existing first-fit partitioning schemes. We also introduce a new first-fit partitioning scheme based on the processing time of tasks, which yields better results than the other two schemes in experimental analysis. We also show that the number of processors required to schedule tasks in P-FRP will be more than or equal to the number of processors required to schedule the same in the preemptive model.

* 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

Partitioned Scheduling of P-FRP in Symmetric Homogenous Multiprocessors*

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. Priority-based FRP (P-FRP) has recently been introduced as a FRP formalism that guarantees real-time response. P-FRP guarantees that when a higher priority task is released, the system will immediately preempt any executing lower-priority tasks. To maintain guarantees of state-less execution offered by a purely functional model of programming, P-FRP implements a transactional nature of execution. Each higher priority task in P-FRP can abort a lower priority task forcing it to restart. Existing work on partitioning tasks in multi-processor systems have been focused on the classical preemptive model‡ of execution. However, due do its transactional nature of execution, the schedulability tests used in the partitioning algorithms for the preemptive model, cannot be applied 'as is' to the P-FRP execution model. While multiprocessor response time analysis of P-FRP has been done in previous work, partitioning schemes for tasks in multi-processor systems have not been presented yet. In this paper, we present an exact schedulability test for P-FRP and use it in two existing first-fit partitioning schemes. We also introduce a new first-fit partitioning scheme based on the processing time of tasks, which yields better results than the other two schemes in experimental analysis. We show that the number of processors required to schedule tasks in P-FRP are more than or equal to the number of processors required to schedule the same in the preemptive model.

Index Terms

Multi-processor scheduling, Response Time Analysis, Real-time System, Functional Programming

I. Introduction

Reactive programming is a paradigm where program variables dependent on external input are automatically updated with any change in input. Functional Reactive Paradigm (FRP) [39], is a declarative programming language for modeling and implementing reactive systems. It has been used for a wide range of applications, notably, graphics [17], robotics [32], and vision [33]. FRP captures both the continuous and discrete aspects of a hybrid system using the notions of behavior and event, respectively. FRP is implemented as a domain-specific language in Haskell [20], and benefits from the wealth of abstractions provided in this language. However, Haskell provides no real-time guarantees, and therefore neither does FRP.

To address this limitation, resource-bounded variants of FRP were studied [24],[37],[38]. It was shown that a variant called priority-based FRP (P-FRP) [24] combines both the semantic properties for FRP, guarantees resource boundedness, and supports assigning different priorities to different events.

* 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

In P-FRP, higher priority events can preempt lower-priority ones. However, a requirement [36] in the functional programming model is that the state of the system cannot be changed, and no function can have side effects. To maintain this guarantee of 'state-less' execution, the functional programming paradigm requires the execution of a function to continue uninterrupted. To comply with this requirement, as well as allow preemption of lower priority events, P-FRP implements a multi-version commit 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 (or *task*), 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.

The benefits of using the functional programming over the imperative programming style (C++, Java, Ada etc.), are several. The functional programming paradigm allows the programmer to intuitively describe safety critical behaviors of the system, thus lowering the chance of introducing bugs in the design phase. Its state-less nature of execution does not require use of synchronization primitives, like mutexes and semaphores, reducing the complexity in programming. This makes the functional programming model ideal for parallel computing, since data sharing conflicts [35], an important consideration in the preemptive execution model, are easily avoided. Hence, P-FRP presents an alternate and promising approach for parallel programming in multi-processor and multi-core systems.

With the availability of low-cost multi-core processors and single boards with multiple processors, more real-time and embedded systems are being implemented as multi-core or multi-processor based. While multiple cores / processors increase the throughput of the system, in real-time implementations where tasks have to complete within deterministic bounds, the assignment of tasks to each individual core/processors has to be carefully determined. Presence of multiple processors also gives fail-safe redundancy, a vital requirement in safety critical systems.

Multi-processor systems can generally be divided into two groups, based on the nature of processors. In a *homogenous* system, all processors are of the same type and execute at the same speed. A *heterogeneous* multi-processor system can have processors of different types which may be running at different speed. Furthermore, *symmetric* multi-processor systems share a common memory bus, while in *asymmetric* multiprocessor systems each processor has access to an independent memory bus. Scheduling algorithms for multi-processor systems can be divided into two distinct categories. *Partitioned* scheduling assigns a processor to every task and all jobs of that task will run on that specific processor. *Global* scheduling, on the other hand, allows tasks to migrate among processors during run time, and hence task assignment to processors is dynamic in nature, as compared to fixed processor assignment in partitioned scheduling. Both these algorithms can be run *off-line* before the system is started if all task parameters are known *a priori*, or *on-line* while the system is running.

In this work, we study off-line partitioning of tasks in symmetric multi-processor system. Our objective is to find the minimum number of processors required to feasibly schedule the tasks in a given P-FRP task set. A feasible schedule is one in which task execution follows a strict priority order, and no task in the system misses its deadline as long as the system is in operation.

Priorities to tasks can be assigned in a static and dynamic way, and the current implementation of P-FRP only allows static priority assignment. Unlike dynamic priority assignment, in fixed priority scheduling, priorities to tasks in the system are assignment off-line and remain fixed as long as the system is running. Common dynamic priority assignment policies are the earliest-deadline-first (EDF) [26] and least-laxity-first (LLF) [28]. The most common fixed priority assignment scheme is the rate-monotonic (RM) priority assignment, which Liu and Layland [26] have shown to be an optimal priority assignment in the preemptive execution model. The optimality of the RM-priority assignment is derived the fact that, if a task set is schedulable by any other priority assignment then it is also schedulable by RM priority assignment. However, Leung and Whitehead [25] showed that the optimality for RM priority assignment is valid only for a *synchronous* (release offset of all tasks is the same) release of tasks.

Several important algorithms have been proposed for the partitioned allocation of tasks in symmetric homogeneous multiprocessor systems. Dhall and Liu [16] have given the Rate-Monotonic-First-Fit-Scheduling (RMFFS) and Rate-Monotonic-Next-Fit-Scheduling (RMNFS) algorithms which combine bin-packing and rate-

monotonic scheduling to assign tasks to processors. Davari and Dhall [13],[14] have given the First-Fit-Decreasing-Utilization-Factor (FFDUF) and the NEXT-FIT-M algorithms. Oh and Son [30] have given the Rate-Monotonic-First-Fit-Decreasing-Utilization (RMFFDU) which uses a different rate-monotonic schedulability test and is an improvement over the RMFFS and RMNF algorithms.

One common aspect of these algorithms is that they use a sorting order for the first-fit scheme. The sorting is performed on the basis of arrival rates or utilization ratios [26]. Another common aspect of these algorithms is that the schedulability test for tasks assigned to a processor, is performed based on criterion defined for the RM scheduling policy. While this criterion is correct in the preemptive execution model, due to a different execution model, the RM-priority assignment is not the optimal priority assignment for P-FRP (see example in Section 3). Hence, none of the first-fit algorithms that have been presented so far are guaranteed to provide correct results in P-FRP. In this paper, we modify existing first-fit based algorithms to use an exact schedulability test designed for P-FRP. We also present a new first-fit criterion based on the processing time of tasks, which is well suited for the P-FRP execution model.

Symmetric multiprocessing for P-FRP was first discussed in [12]. The work in [12] only deals with response time analysis under a pre-assigned partitioning of tasks. However, response time analysis can be suitably performed only when a good partitioning of tasks among multiple processors is known. This work analyzes such partitioning schemes and suggests the most suitable partitioning for P-FRP. Knowledge of such partitioning schemes is vital to any multi-processor implementations of P-FRP. Our work also benefits real-time research beyond the functional programming model of P-FRP. After modifications, methods presented in this work can be applied for multi-processor partitioning in systems with similar abort-restart execution models as transactional memory [21], lock-free execution [2] and real-time databases [7].

I.I Contributions

We first present basic uniprocessor schedulability conditions (Section III) and then propose an exact schedulability test for P-FRP (Section IV). For any first-fit scheme such an exact test is vital for determining the schedulability of some task assignment to processors. We present modified forms of the first-fit decreasing rate (Section V) and first-fit-decreasing utilization factor (Section VI) algorithms. Since, every preempted task is aborted, additional costs are induced on the response time of lower priority tasks. The abort cost is related to the processing time of a task, hence a first-fit partitioning scheme based on the processing time of tasks is also developed (Section VII). Lastly, we present a method to compute the optimal partition for any given task set (Section VIII). These algorithms can be applied for static partitioning to any real-time multiprocessor implementations of P-FRP.

The relative benefits of each of these algorithms have determined using rigorous experimental analysis (Section IX). We have computed the processor requirements under each partitioning scheme for unique task sets having 6,8 and 10 tasks and shown comparisons between various parameters. Processor requirements between the P-FRP and preemptive execution models have also been compared. Apart from proving the correctness of our algorithms, these results provide data to engineers on the relative merits of each partitioning scheme. We conclude this paper by reviewing related work (Section X) and a reflection on these results (Section XI).

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 concept of arrival rate and their processing time; the concept of a time interval and task jobs therein. 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
- In a multi-processor system let $\Gamma_n[m]$ represent the set of n tasks that are statically assigned to execute in processor m
- The **priority** of $\tau_k \in \Gamma_n$ is the integer pr_k. If, $pr_j > pr_k$ then τ_j has a higher priority than τ_k
- T_k is the **time period** between two successive jobs of τ_k
- C_k is the **worst-case execution time (WCET)** 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, $P_k = t_{copy}(k) + C_k + t_{restore}(k)$
- $[t_1, t_2)$ represents a **time interval** such that: $\forall t \in [t_1, t_2)\ \ t_1 \le t < t_2 \wedge t_1 \ne t_2$, t_1 and t_2 are absolute times
- $R_{k,m}$ 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, $R_{k,m} = \Phi_k + (m-1)\cdot T_k$
- A **level-*k* idle period** is defined as a point in time, t in which no task having a priority of pr_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 period
- The **threshold** of the k-gap $[t_1, t_2)$ is time t_1
- D_k is the **relative deadline** of τ_k. After its release a job of τ_k should complete execution within D_k time units, otherwise the task will have a **deadline miss.** For this study, $D_k = T_k$.
- The **total utilization factor** (U) of a task set is the sum of ratios of processing time to arrival periods of every task. It is represented by U in this paper. Hence, $U = \sum_{i=1}^{n} \frac{P_i}{T_i}$
- A **feasibility interval** is the time interval $[t_H, t_H + H)$ such that if all tasks are schedulable in $[t_H, t_H + H)$ then the tasks will also be schedulable in the time interval $[0, Z)$: $Z \to \infty$. H is the length of the feasibility interval and t_H is its start time
- PA_k represents the processor statically assigned to process task τ_k, by some partitioning algorithm
- An **optimal partition** is one, which requires the least number of processors. No other partitioning scheme can exist which can feasibly schedule a P-FRP task set in lesser number of processors than that required in an optimal partition
- **Interference** on a task τ_k is the process where the execution of τ_k is interrupted by the release of a higher priority task.

II.II Execution Model and Assumptions

In this study, we assume a symmetric homogenous multi-processor system with a single time clock. The P-FRP tasks running in these processors do not have precedence constraints. The current implementation of P-FRP uses fixed priority scheduling, hence all tasks are assigned a static priority before execution.

In the P-FRP execution model, when job of a higher priority task 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 execution. When some task is released, it enters an execution 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 jobs of the same task can be present in the queue at the same time. This requires a task to complete execution before the release of its next job. To maintain

Figure 1(a): τ_1 has a deadline miss at time 80 when the $pr_3 > pr_2 > pr_1$. τ_1 ,τ_2 and τ_3 are represented by T1, T2 and T3 respectively

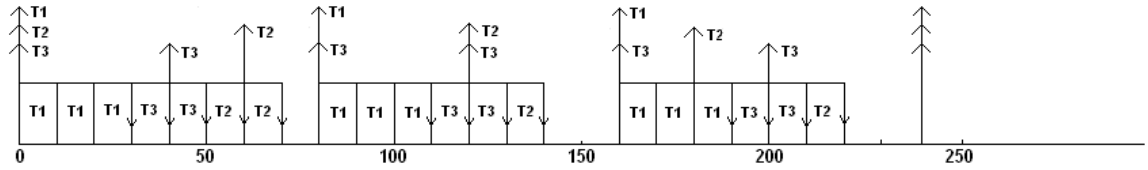

Figure 1(b): All tasks are schedulable in the feasibility interval [0,240) if the priority order is changed to $pr_1 > pr_3 > pr_2$

this requirement, we assume a hard real-time system with task processing deadline equal to the minimal necessary wait. Hence, $D_k = T_k$.

Once a task τ_i enters Q, two situations are possible. If a task of lower priority than τ_i is executing, it will be immediately preempted and τ_i will start execution. If a task of higher priority than τ_i is executing, then τ_i will wait in the Q and start execution 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 execution 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 between copy and restore operations, the scratch state is simply discarded. The time to discard the state of an aborted task is minimal has been ignored in this study. The context-switch between tasks only involves a state copy operation for the task that will be commencing execution. The time taken for copy (t_{copy}) and restore ($t_{restore}$) operations of τ_k is part of the processing time of the task, P_k. For this study, the values of $t_{copy}(k)$ and $t_{restore}(k)$ for all tasks are kept same and equal to a single time unit of execution. Hence,

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

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 better accuracy of results, in ongoing work we will be 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) [26] of the task being analyzed. In their seminal work, Liu and Layland [26] 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, tasks τ_i and higher priority tasks are released synchronously. In P-FRP, a synchronous release of tasks does not lead to the WCRT, for all cases. This is proven by an example given in [8], where the WCRT is caused by an *asynchronous* (tasks have different release offsets) release of higher priority tasks.

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 actual response time under each of the possible release offset combina-

tions have to be computed using the gap-enumeration algorithm presented in [8]. Finally, the highest value of the response time computed for each release offset will be the WCRT for the task.

In this paper, we have only considered offset-free systems, therefore all tasks are assumed to be released synchronously. Future work will involve analyzing worst-case release scenarios in offset-free systems.

III. P-FRP Scheduling Characteristics in a Uniprocessor

Partitioning among multiple processors is based on the scheduling requirements in a uniprocessor. In this section, apart from stating a theorem for the feasibility interval, we show that the rate-monotonic priority assignment is not optimal in P-FRP, and prove that all necessary scheduling conditions for the preemptive execution model are also necessary conditions in P-FRP. We also state a schedulability property of P-FRP tasks sets, based on which we derive a P-FRP exact schedulability.

Theorem 3.1 [9]: *If tasks in Γ_n are released synchronously , then the feasibility interval for Γ_n is* [0,*L*),*where L is the Least Common Multiple (LCM) of all task periods in Γ_n.*

Lemma 3.2: *In P-FRP the rate-monotonic scheduling policy is not an optimal priority assignment with synchronous release of tasks.*

Proof. If we can give a P-FRP task set which is not schedulable using the RM priority assignment, but is schedulable by a priority assignment which is not RM, it is sufficient to prove this lemma. Consider the following task set:

Task	*pr*	*P*	*T*
τ_1	1	30	80
τ_2	2	10	60
τ_3	3	10	40

The priority assignment is RM-based with τ_3 having the highest arrival rate hence, the highest priority. In this scheduling policy, if all tasks are released synchronously, the first job of τ_1 is unable to complete processing before its second job at time 80 (Figure *1(a)*). If the priority order is changed, as shown below:

Task	*pr*	*P*	*T*
τ_2	1	10	60
τ_3	2	10	40
τ_1	3	30	80

Then, jobs of all tasks will be able to complete processing in the feasibility interval [0,240) of Γ_n (Figure *1(b)*). □

Lemma 3.3: *If a P-FRP task set Γ_n is schedulable under some priority assignment, then Γ_n is guaranteed to be schedulable for the same priority assignment in the preemptive execution model.*

Proof. The response time of the highest priority task in P-FRP and the classical model is the same. Higher priority tasks can cause interference in the processing of lower priority tasks. In P-FRP, this interference leads to abort, which puts a higher cost on the processing of lower priority tasks as compared to the preemptive model. Two situations are possible:

No interference from higher priority tasks: The difference in response time between P-FRP and preemptive model is due to the abort of lower priority tasks which is caused by interference from higher priority tasks. Hence, if there is no interference, there will be no aborts and response time for all tasks in both execution models

will be same. Therefore, in this case if the task set is schedulable in P-FRP, it will also be schedulable in the preemptive model.

Interference from higher priority tasks: Consider, $\Gamma_2 = \{\tau_i, \tau_j\}$ and $pr_i > pr_j$:

Let τ_j be released at absolute time t_a and execute for h time: $t_{copy}(j) \le h \le t_{copy}(j) + C_j$, after which it is aborted by the release of a job of τ_i. τ_j will re-execute after τ_i has completed processing at absolute time t_a+h+P_i. τ_j will take another P_j time to complete processing and will finish at absolute time $t_a+h+P_i+P_j$. Hence, its response time is: $h+P_i+P_j$. For processing in the classical model, the response time of τ_j will be $h+P_i+P_j-h = P_i+P_j$. Hence, after interference from higher priority tasks, the response time of lower priority tasks in P-FRP will always be more than the preemptive model. Therefore, in this case, if the task is schedulable in P-FRP, it will also be schedulable in the preemptive model. □

Lemma 3.4: *Schedulability conditions which are necessary for preemptive model are also necessary conditions for schedulability in P-FRP.*

Proof. As shown in lemma 3.3, a task set schedulable in P-FRP is guaranteed to be schedulable in the preemptive execution model. Every schedulable task set in the preemptive model will satisfy the necessary conditions of this model. Since schedulability in the preemptive model is a requirement for the schedulability in P-FRP, every necessary condition in the preemptive model is also a necessary condition for P-FRP.

Note, however that sufficient schedulability conditions for the preemptive model are not guaranteed to be sufficient conditions for P-FRP. If a sufficient schedulability condition is satisfied, it guarantees the schedulability of a task set. Since, a task set which is schedulable in the preemptive model, can be unschedulable in P-FRP, sufficient schedulability conditions for the preemptive model cannot be used to determine schedulability in P-FRP. □

Theorem 3.5: *For task set Γ_n, a necessary condition for scheduling tasks in P-FRP is that the combined utilization factor (U) of all tasks in Γ_n should be less than or equal to unity. Or,*

$$\text{If } U = \sum_{i=1}^{n} \frac{P_i}{T_i}, \text{ then } U \le 1.$$

Proof. This is a necessary condition for fixed priority scheduling in the preemptive model, as shown in [26]. Since, P-FRP will satisfy all necessary conditions true for the preemptive model as postulated in lemma 3.4, this condition will be satisfied for any schedulable for P-FRP task set. This condition is necessary but not sufficient in the sense that every schedulable P-FRP task set will satisfy this condition, but the satisfiability of this condition alone, does not guarantee the schedulability of a task set. □

Lemma 3.6: *For a task τ_j to be schedulable, one j-gap of length greater than or equal to P_j should exist between any two successive jobs of τ_j.*

Proof. In the P-FRP execution model, a task τ_j can complete processing only in a *j*-gap. Assuming a task τ_j is released at *t*, a *j*-gap should be available before time $t + T_j$, when the next job of τ_j is released. If no *j*-gap is available in the interval $[t, t + T_j)$, then τ_j will have a deadline miss and will not be schedulable. □

Corollary 3.6.1: *For schedulability of task set Γ_n, there should be a j-gap larger than or equal to the processing time of τ_j, available between successive jobs of τ_j for $\forall \tau_j \in \Gamma_n$.*

Proof. For the task set to schedulable, lemma 3.6 should be satisfied for every task that is a member of Γ_n. □

Corollary 3.6.2: *Let a time duration H and time t exist such that the task processing pattern of Γ_n repeats itself in time intervals [t,t+H), [t+H,t+2·H), [t,t+3·H)... Then an exact condition for the schedulability of Γ_n, is that for*

every $\tau_j \in \Gamma_n$ *, there should be a gap larger than the processing time available between all jobs of* τ_j *in the time interval [t, t+H).*

Proof. For the task set to schedulable, the condition given in lemma 3.6 should be satisfied for every task in Γ_n, as given by corollary 3.6.2. If all the tasks are schedulable in [*t, t+H*), they will be schedulable in all the other time intervals following [*t, t+H*). This is an exact schedulability condition for a P-FRP task set. Such a test is sufficient in the sense that if this test is satisfied, the task set is guaranteed to be schedulable. This condition is also necessary in the sense that it will be satisfied by every schedulable P-FRP task set. □

IV. An Exact Schedulability Test Algorithm for P-FRP

In the previous section, we have shown that the rate-monotonic priority assignment is not guaranteed to be the optimal priority assignment for P-FRP. Hence, the condition used to check the schedulability of tasks assigned processors in previous works [16], [13],[14],[30], cannot be used to check schedulability in P-FRP. Till now, a closed form sufficient schedulability test condition for P-FRP is unknown. Hence, using an algorithm based approach to evaluate the exact schedulability condition, presented in corollary 3.6.2, is the only way to ascertain schedulability of a P-FRP task set.

We present an algorithm that can determine the schedulability of tasks assigned to a processor, based on the length of *k*-gaps left after the execution of higher priority tasks. This algorithm is based on the gap-enumeration method [8], which has been previously developed for P-FRP.

Some new definitions used in the definition of this algorithm are:

- A **gap set** $\sigma_k([t_1,t_2))$ contains all the unique *k*-gaps present in the time interval $[t_1,t_2)$. The gaps present in $\sigma_k([t_1,t_2))$ are also disjoint:

  for any two gaps $[t_{x1},t_{y1}), [t_{x2},t_{y2}) \in \sigma_k([t_1,t_2))$, if $t \in [t_{x1},t_{y1})$ then $t \notin [t_{x2},t_{y2})$

- $|\sigma_k([t_1,t_2))|$ represents the number of *k*-gaps present in the gap set $\sigma_k([t_1,t_2))$

- The **gap-transformation function** $\lambda(\sigma_k([t_1,t_2)), \Gamma_n)$ 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:
  $\sigma_{k-1}([t_1,t_2)) = \lambda(\sigma_k([t_1,t_2)))$

- The **gap-search function** $\mu(\sigma_k([t_1,t_2)), P_k)$ takes as input the gap set $\sigma_k([t_1,t_2))$ and P_k, and returns the earliest *k*-gap larger than or equal to P_k present in σ_k :

  $[t_{x1},t_{y1}) = \mu(\sigma_k([t_1,t_2)), P_k)$, such that
  $t_{y1} - t_{x1} \geq P_k \wedge \nexists\ [t_x,t_y) \in \sigma_k([t_1,t_2)) \wedge t_y - t_x > P_k \wedge t_x < t_{x1}$

If the gap search function returns a gap with threshold less than 0 then a *k*-gap larger than P_k does not exist in $\sigma_k([t_1,t_2))$. The basic gap-enumeration algorithm to determine the response-time of τ_j (RT_j) as presented in [8] is:

Gap Enumeration Algorithm

```
1.  input: Γn, τj
2.  output: RTj
3.  σn([0, Tj)) ← {[0,Tj)}
4.
5.  loop task i ← n to j+1
6.        σi-1([0, Tj))←λ (σi([0, Tj)),Γn)
7.        if(|σi-1([0, Tj))| = 0) return -1
```

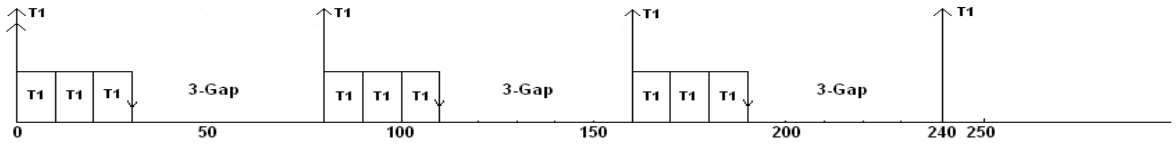

Figure 2(a): Compute 3-gaps in which τ_3 can execute after accounting for execution of τ_1 in the feasibility interval [0,240)

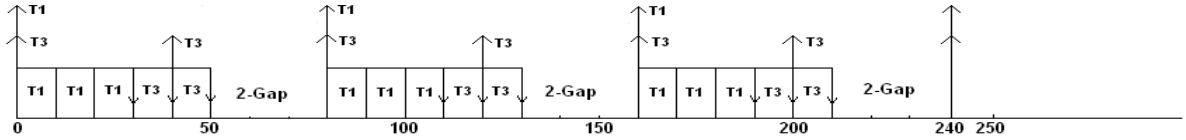

Figure 2(b): Compute 2-gaps in which τ_3 can execute after accounting for execution of τ_1 in the feasibility interval [0,240)

---

8. end loop
9. $[t_1,t_2) \leftarrow \mu(\sigma_j([0, T_j)), P_j)$
10. if($t_1 < 0$) return -1
11. $RT_j = t_1 + P_j$
12. return RT_j

This algorithm can be modified to determine the schedulability of a task set, using corollary 3.6.2. In corollary 3.6.2, the time duration $[t,t+H)$ is the feasibility interval of Γ_n, which for a synchronous release is $[0, L)$, (L is the LCM of all period in Γ_n).

First, we find all the 1-gaps that will be available for processing multiple jobs of the lowest priority task τ_1 in the feasibility interval of the task set. Hence, replacing T_j by L, in the original gap analysis algorithm will give us this information. Then, the gap search method $\mu([0, L), P_j)$ will analyze the gaps and insure that there is a gap of at least length P_1 between each job of τ_1. If the gap search method returns *false*, it implies a gap of minimum P_1 length does not exist between some job of τ_1 in the time interval $[0, L)$, making the task set unschedulable. After making these modifications to the gap enumeration algorithm, the exact schedulability test algorithm is derived:

Algorithm 5.1: Exact Schedulability Test Algorithm

1. input: Γ_n, $[0, L)$
2. output: True/False depending on schedulability
3. if(n==1) return true
4. $\sigma_n([0, L)) \leftarrow \{[0, L)\}$

5. loop task $i \leftarrow n$ to 2
6. $\quad\sigma_{i-1}([0, L)) \leftarrow \lambda(\sigma_i([0, L)), \Gamma_n)$
7. $\quad$if($|\sigma_{i-1}([0, L))| = 0$) return false
8. end loop
9. status $\leftarrow \mu(\sigma_1([0, L)), P_1)$
10. return status

The algorithm for the modified gap-search function is:

Algorithm 5.2: Gap-Search Function

1. input: $\sigma_1([0, L)), P_1$
2. output: True/False depending on schedulability
3. $jobs_1 \leftarrow 0$
4. startTime $\leftarrow \Phi_k$
5. loop for each gap $[t_1,t_2)$ in $\sigma_1([0, T_j))$
6. $\quad$if($t_2 - t_1 \geq P_1$)
7. $\quad\quad$if($t_1 + P_1 \leq$ startTime $+ T_1$)
8. $\quad\quad\quad jobs_1 \leftarrow jobs_1 + 1$
9. $\quad\quad\quad$startTime$\leftarrow$ startTime $+ T_1$
10. $\quad$else if ($t_1 + P_1 >$ startTime $+ T_1$)
11. $\quad\quad$return false

12. end loop
13. if $jobs_1$ = jobs([0, L),1) return true
14. else return false

Definition: The function $\Omega(\Gamma_n, [0, T_j))$, is defined as an exact schedulability test and represents algorithm 5.1. Only if, $\Omega(\Gamma_n, [0, T_j))$ = true, will Γ_n be schedulable in P-FRP.

IV.I Example

We illustrate this method using the example used in Lemma 3.2. We have already shown the task set to be schedulable in the following priority order:

Task	*pr*	*P*	*T*
τ_2	1	10	60
τ_3	2	10	40
τ_1	3	30	80

We first find the 3-gaps in which jobs of task with the 2nd highest priority (τ_3) can execute in the feasibility interval of [0,240). These 3-gaps are determined after accounting for the execution of all jobs of the highest priority task (τ_1). There will be 3 jobs of τ_1 in the interval [0,240), creating 3-gaps in the intervals [30,80), [110,160) and [190,240) (*Figure 2(a)*). After task τ_3 is run, 2-gaps in intervals [50,80), [130,160) and [210,240) are created (*Figure 2(b)*). We then search these 2-gaps to make sure a 2-gap of length more than or equal to P_2 is present between the 4 jobs of τ_2 that are released in the interval [0,240). As seen in *Figure 1(a)*, such 2-gaps exist for all 4 jobs of τ_2, hence the task set is schedulable in its feasibility interval.

IV.II Time Complexity

The complexity of the exact schedulability test (algorithm 5.1) is based on the combined time complexity for gap analysis and gap search method. The complexity of the gap analysis method as given in [8], is bounded by:

$$O((n\text{-}j) \cdot |\sigma_i([0, T_j))| \cdot \text{jobs}(i, [0, T_j)) \cdot \log(!2 \cdot 2 \cdot |\sigma_i([0, T_j))|)).$$

Where, i: $T_i = \min(T_{n-j}, T_{n-j+1} \ldots T_n)$.

Since, in algorithm 5.1, $T_j = L$, and j =1, the complexity for gap analysis is:
$O((n\text{-}1) \cdot |\sigma_i([0, L))| \cdot jobs_i(i, [0, L)) \cdot \log(!2 \cdot 2 \cdot |\sigma_i([0, L))|))$

Here, τ_i is the task with the highest arrival rate among higher priority tasks, and $jobs_i$ represent the number of jobs of τ_i in [0,L). In the worst-case, the gap search function will iterate for all gaps present in $\sigma_1([0, L))$, hence, its complexity is bounded by $O(|\sigma_1([0, L))|)$. The total complexity is the sum of the worst-case time for gap-analysis and gap-search. Since the complexity for gap-analysis is the dominating term, the time complexity of the sufficient schedulability test represented by $O(\Omega(\Gamma_n, L))$, is:

$$O((n\text{-}1) \cdot |\sigma_i([0, L))| \cdot jobs_i \cdot \log(!2 \cdot 2 \cdot |\sigma_i([0, L))|)).$$

The sufficient schedulability test is now used in the partitioning algorithms described from the next section.

V. P-FRP First-Fit-Decreasing Rate

Static partitioning of tasks between multiple processors represents the classical bin-packing problem. In bin-packing, items of different sizes have to be packed in a finite number of bins. For our partitioning problem, items represent the tasks while the bins are the processors. Since the bin-packing algorithm is proven to be computa-

tionally NP-hard, several heuristics are available to derive an approximate solution. Due to its simplicity, the First-Fit (FF) method is a popular greedy approximation heuristic for the bin-packing problem, and has a tight bound of 1.7 [23]. In the FF algorithms for partitioning tasks in multi-processors, a task is assigned to the first processor in which is can be feasibly scheduled. Each of the processors which already have tasks assigned to them, are searched in sequential order and checked if the previous tasks can be feasibly scheduled along with the new task. In previous works, it has been proven that the order in which tasks are sorted before being assigned to processors, affect the efficiency of the algorithm. We have presented three different FF based algorithms in this paper based on different sorting criterion.

The first criterion is decreasing rate, and a FF-based algorithm for RM-scheduling using this criterion was presented in [16]. In the algorithm given in this section, we use the FF heuristic with the decreasing rate criterion, along with the P-FRP exact schedulability test. The algorithm returns the minimum number of processors required to partition the tasks in such a way that all tasks are schedulable. Since the exact schedulability test is computationally intensive, we check the combined utilization factor of tasks in processor j as a necessary schedulability condition (line 7). If the combined utilization factor is greater than 1, then the task set $\Gamma_n[j]$ is guaranteed to be unschedulable and the exact schedulability test (line 8) does not need to be executed. Other necessary schedulability tests for the preemptive model can also be used (as per lemma 3.4), to minimize the possibility of executing the P-FRP exact schedulability test for unschedulable tasks.

Once all tasks have been assigned, the number of processors required to schedule all tasks is returned (line 11). $L(\Gamma_n[j])$ represents the LCM of all tasks periods present in task set $\Gamma_n[j]$.

P-FRP FF-Decreasing-Rate Algorithm (PFFDR)

1. input: Γ_n
2. output: m
3. Sort tasks in Γ_n in order of decreasing rate such that τ_1 becomes the task with the highest rate and τ_n the lowest
4. $m \leftarrow 1$: $i \leftarrow 1$
5. $j \leftarrow 1$
6. Add τ_i to $\Gamma_n[j]$
7. If $\sum_{\forall i \in \Gamma_n[j]} \frac{P_i}{T_i} > 1$: $j \leftarrow j+1$: goto step 6
8. If $\Omega(\Gamma_n[j], [0, L(\Gamma_n[j]))$ is true: $PA_i \leftarrow j$
9. else: $j \leftarrow j+1$: goto step 6
10. if($j>m$): $m \leftarrow j$
11. if ($i==n$): return m
12. else: $i \leftarrow i+1$: goto step 5

V.I Time Complexity

We present an analysis of the time-complexity of the PPFDR algorithm. The complexity of sorting all the items will take $O(n \log n)$ time. If every task takes one processor to run, then the algorithm will check if a task can be assigned to one of the processors which already have tasks assigned to them. To assign the 1st task, the algorithm will run for 1 step, for the 2nd task, 2 steps and so on. Hence, the total number of steps the algorithm will run in the worst-case is:

$$1+2+\ldots+n = \frac{n}{2}(n+1).$$

This term is bounded by $O(n^2)$. Since $O(n^2)$ is the dominant term, the complexity of the FF heuristic is bounded by $O(n^2)$. Since there is only one task, the gap-enumeration will return true without going through any computation steps.

However, in P-FRP's case the worst-case for the PFFDR algorithm will be when all tasks are assigned to the same processor, due to higher costs of the P-FRP schedulability test. If all tasks are assigned to the same processor then the FF heuristic will compute in $O(n)$ time, but the time to run $\Omega(\Gamma_n, L(\Gamma_n[1]))$ during each search of processor for a task is bounded by:

$O(\Omega(\Gamma_n, [0, L))$ (since, $L(\Gamma_n[1] = L)$).

Since, the P-FRP schedulability test will be run n times, the worst-case complexity for the PFFDR algorithm is:
$O(n \cdot O(\Omega(\Gamma_n, [0, L))$.

VI. P-FRP First-Fit-Decreasing Utilization Factor

The P-FRP first fit decreasing utilization factor algorithm (PFFDUF), sorts tasks based on their utilization factors. A similar sorting criterion has been used in [14], [30]. This algorithm is same as PFFDR, with line 3 modified to:

Change in PFFDR for PFFDUF

3. Sort tasks in Γ_n in order of decreasing utilization factor such that τ_1 becomes the task with the highest utilization factor and τ_n the lowest.

The time complexity of this algorithm is the same as that of PFFDR.

VII. P-FRP First-Fit-Decreasing Processing Time

Due to abort, additional delay classified as 'abort cost' is induced on the response time of a preempted lower priority task. This abort cost is dependent on the processing time of the aborted task. The following lemmas define some properties of abort cost.

Lemma 7.1: *The upper bound of abort cost induced on a lower priority task equals the processing time of the lower priority task plus the time to copy the state of the task. The lower bound is the time to copy the state of the task.*

Proof. Consider, $\Gamma_2 = \{\tau_i, \tau_j\}$ and $pr_i > pr_j$. Let τ_j be released at time t and is processed for h time, after which a job of τ_i is released. τ_j will restart processing after τ_i has completed, and its response time will be $t_a + h + P_i + P_j$. The only variable in the response time of τ_j is h. If, $h > t_{copy}(j) + C_j$ then τ_j has completed processing, and is committing its results. As per the P-FRP execution model, τ_j cannot be preempted at this stage, and τ_i will start only when τ_j has completed it processing. If $h < t_{copy}(j)$, then τ_j is copying its state and cannot be preempted. However, as soon as the copy is over, τ_j will be preempted and τ_i will commence processing. Hence, τ_j will have to be processed for $t_{copy}(j)$ time, before it can be preempted, and can be processed for maximum $t_{copy}(j)+C_j$ time, after which preemption is not possible. Therefore, to induce an abort cost on τ_j, h will lie in the range $[t_{copy}(j), t_{copy}(j)+C_j]$. The limits of this range are the lower and upper bounds of the abort cost of τ_j. □

Lemma 7.2: *If time to copy and restore is the same then the task with lower processing time will have a lower maximum abort cost.*

Proof. Consider, $\Gamma_2 = \{\tau_i, \tau_j\}$ and $P_i > P_j$. Since $t_{copy}(i) = t_{copy}(j)$ and $t_{restore}(i) = t_{restore}(j)$, $\Rightarrow C_i > C_j$. The maximum abort cost that can be induced on τ_i is $t_{copy}(i)+C_j$ while on τ_j is $t_{copy}(j)+C_j$. Clearly,
$t_{copy}(j) + C_j < t_{copy}(i)+C_j$. □

From lemmas 7.1 and 7.2, we can deduce that the processing time of tasks assigned to a processor is an important factor in determining the response time of lower priority tasks, and thereby, the schedulability of the tasks assigned to that processor. If we sort the tasks based on their processing times, there is a higher probability of tasks being assigned in such a way, that those with larger processing times (hence, higher abort costs), will be

placed in a processor which has lesser number of tasks. This reduce the chance of preemption of tasks with higher processing times.

We present an algorithm where the FF heuristic is applied to tasks sorted in order of their processing time. This algorithm is same as PFFDR, with line 3 modified to:

Change in PFFDR for PFFDPT

3. Sort tasks in Γ_n in order of decreasing processing time such that τ_1 becomes the task with the highest processing time and τ_n the lowest

The time complexity of this algorithm is the same as for the PFFDR algorithm.

VIII. P-FRP Optimal Partitioning

An optimal partitioning scheme in one which gives requires the minimum number of processors to schedule a P-FRP task set. For a P-FRP task set it is guaranteed that, there is no other partitioning scheme that can result in a lesser number of processors than that given by the optimal partition. We use the optimal partition for performance comparison with the other three partitioning schemes presented in this paper.

To derive an optimum partition for P-FRP tasks we evaluate every possible combination of task assignment, to see which one will result in the least number of processors. For a given number of processors *m*, possible combinations of task partitions can be derived by constructing a B-tree, with every node having *m* children. Each node in the B-tree represents a processor and the level of the node represents the task that is assigned to run on that processor. To find out the minimum number of processors required to schedule task set Γ_n, we start from setting m=1, and increment *m* till some partitioning scheme is available in which Γ_n is schedulable.

Deriving an optimal partitioning for a task set using this method requires evaluation of exponential combinations of task partitions, and therefore, this method is not suitable for practical scenarios.

IX. Experimental Results

We have evaluated the efficiency of the partitioning schemes presented in this paper, using synthetic task sets. We generated 3 groups of 500 tasks sets, with task sets in each group having 6, 8 and 10 tasks. Every task set in a group is unique in the sense that, at least one task is different between two task sets. The processing times of the tasks were selected from the range [5,20], while their arrival rates were selected from [10,40]. All tasks were released synchronously, and no single task has a utilization factor greater than 0.3. By bounding the utilization factors of tasks, we make sure that more than one task can be scheduled in a single processor.

For each of the 3 groups we computed the number of processors required to schedule the task sets under PFFDF, PFFDUF and PFFDPT partitioning schemes. The number of processors required under an optimal partitioning scheme is also derived. We also compute the number of processors required by the PFFDR, PFFDUF, PFFDPT and optimal partitioning schemes under the preemptive execution model.

Figures *3(a), 4(a)* and *5(a)* show the number of processors required under the PFFDR, PFFDUF and PFFDPT partitioning schemes respectively, for 6 tasks. Figures *3(b), 4(b)* and *5(b)* show the difference between number of processors required by PFFDR, PFFDUF and PFFDPT algorithms, respectively as compared to the optimal partitioning. With the P-FRP execution model, the number of task sets whose processor requirements are more than that given by the optimal partitioning for PFFDR, PFFDUF and PFFDPT is 61, 51 and 29 respectively. Figures *3(c), 4(c)* and *5(c)* show the difference in number of processors required by FFDR, FFDUF and FFDPT partitioning under the preemptive execution model. Figures *3(d), 4(d)* and *5(d)* show the difference in processor requirements under optimal partitioning with FFDR, FFDUF and FFDPT in the preemptive execution model. With 6 tasks under the preemptive execution model, all the partitioning schemes required the same number of processors as optimal partitioning.

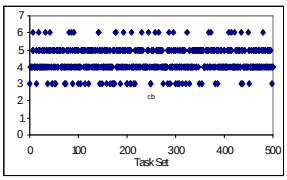

Figure 3 (a): 6 Tasks - # Processors FFDR

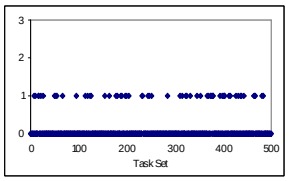

Figure 3 (b): P-FRP - Δ (FFDR, Optimal)

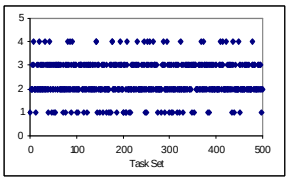

Figure 3 (c): Δ (P-FRP, Preemptive)

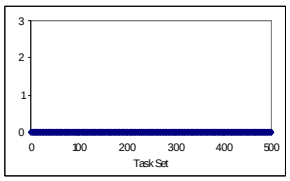

Figure 3 (d): Preemptive - Δ (FFDR, Optimal)

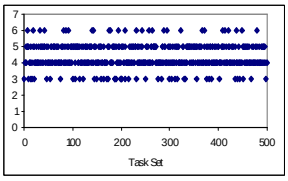

Figure 4 (a): 6 Tasks - # Processors FFDUF

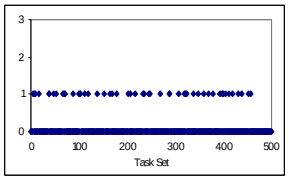

Figure 4 (b): P-FRP: Δ (FFDUF, Optimal)

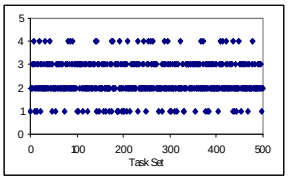

Figure 4 (c): Δ (P-FRP, Preemptive)

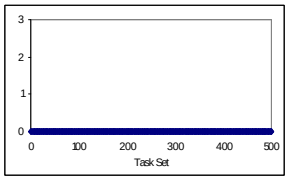

Figure 4 (d): Preemptive - Δ (FFDUF, Optimal)

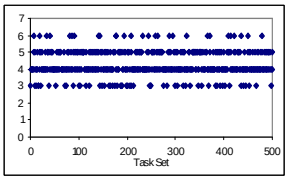

Figure 5 (a): 6 Tasks - # Processors FFDPT

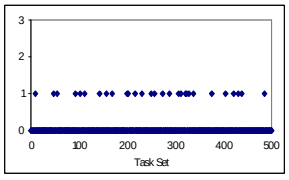

Figure 5 (b): P-FRP: Δ (FFDPT, Optimal)

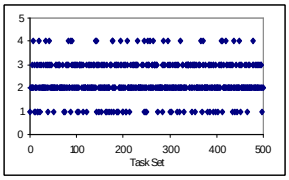

Figure 5 (c): Δ (P-FRP, Preemptive)

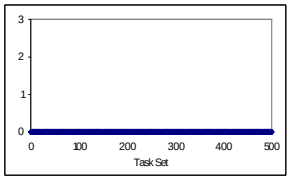

Figure 5 (d): Preemptive:Δ (FFDPT, Optimal)

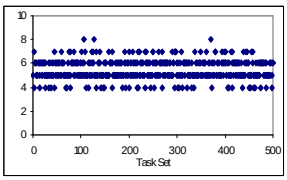

Figure 6 (a): 8 Tasks - # Processors FFDR

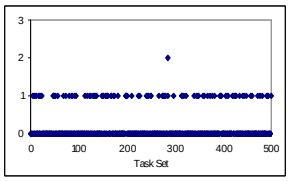

Figure 6 (b): P-FRP: Δ (FFDR, Optimal)

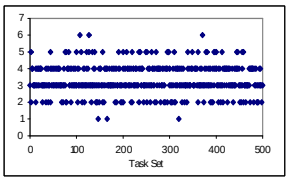

Figure 6 (c): Δ (P-FRP, Preemptive)

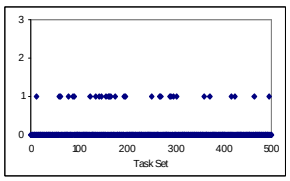

Figure 6 (d): Preemptive: Δ (FFDR, Optimal)

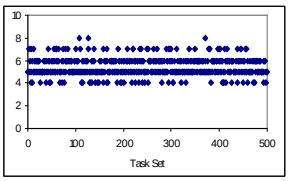

Figure 7 (a): 8 Tasks - # Processors FFDUF

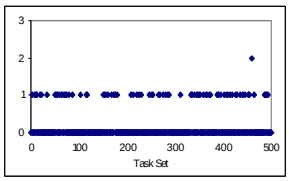

Figure 7 (b): P-FRP: Δ (FFDUF, Optimal)

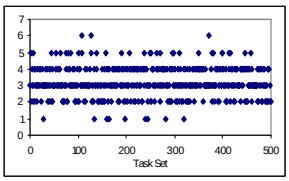

Figure 7 (c): Δ (P-FRP, Preemptive)

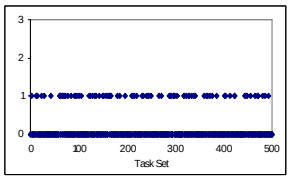

Figure 7 (d): Preemptive: Δ (FFDR, Optimal)

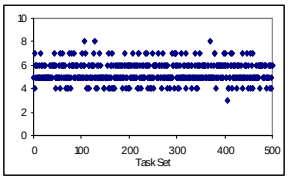

Figure 8 (a): 8 Tasks - # Processors FFDPT

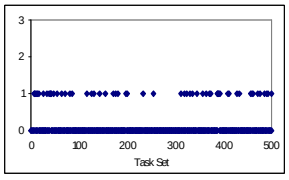

Figure 8 (b): P-FRP: Δ (FFDPT, Optimal)

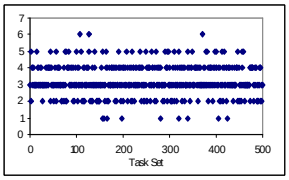

Figure 8 (c): Δ (P-FRP, Preemptive)

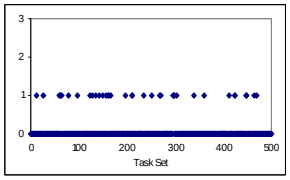

Figure 8 (d): Preemptive: Δ (FFDR, Optimal)

Figures *6(a)-6(d),7(a)-7(d),8(a)-8(d)* contain the same set of data for 8 task-sets. With the P-FRP execution model, the number of task sets whose processor requirements are more than that given by an optimal partitioning for PFFDR, PFFDUF and PFFDPT is 99, 84 and 58 respectively. Under the preemptive execution model, the number of task sets whose processor requirements are more than that given by an optimal partitioning for FFDR, FFDUF and FFDPT is 30, 86 and 36 respectively. Figures *9(a)-9(d),10(a)-10(d),11(a)-11(d)* contain the same set of data for 10 task-sets. With the P-FRP execution model, the number of task sets whose processor requirements are more than that given by an optimal partitioning for PFFDR, PFFDUF and PFFDPT is 177, 168 and 107 respectively. Under the preemptive execution model, the number of task sets whose processor requirements are more than that given by an optimal partitioning for FFDR, FFDUF and FFDPT is 7, 5 and 6 respectively. Clearly, for P-FRP, the FFDPT algorithm performs closest to the optimal partition for maximum number of task sets, relative to the FFDR and FFDUF partitioning algorithms. In the preemptive execution model, the FFDR and FFDUF have better performance than FFDPT. From figures *3(c)-11(c)*, we can deduce that the number of processors re-

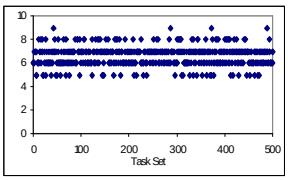

Figure 9 (a): 10 Tasks - # Processors FFDR

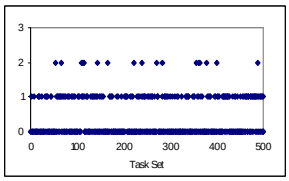

Figure 9 (b): P-FRP - Δ (FFDR, Optimal)

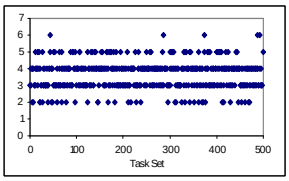

Figure 9 (c): Δ (P-FRP, Preemptive)

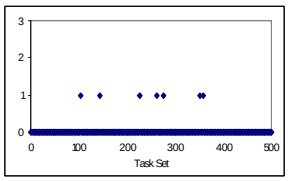

Figure 9 (d): Preemptive: Δ (FFDR, Optimal)

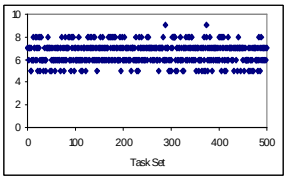

Figure 10 (a): 10 Tasks - # Processors FFDUl

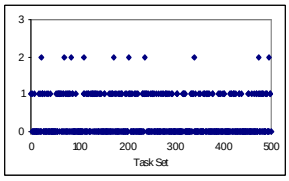

Figure 10 (b): P-FRP - Δ (FFDUF, Optimal)

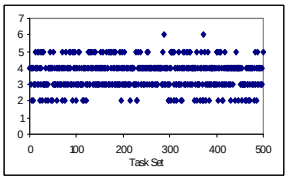

Figure 10 (c): Δ (P-FRP, Preemptive)

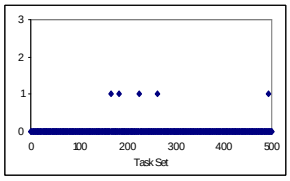

Figure 10 (d): Preemptive: Δ (FFDUF, Optimal)

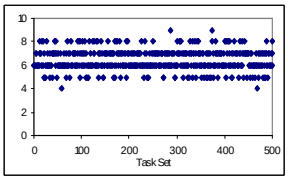

Figure 11 (a): 10 Tasks - # Processors FFDPT

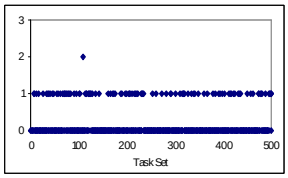

Figure 11 (b): P-FRP - Δ (FFDPT, Optimal)

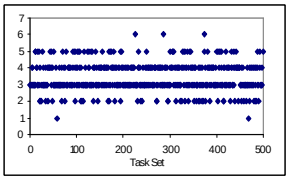

Figure 11 (c): Δ (P-FRP, Preemptive)

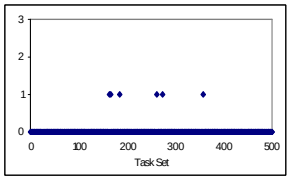

Figure 11 (d): Preemptive: Δ (FFDPT, Optimal)

quired by P-FRP is always greater than or equal to the number of processors required by the same task set in the preemptive model. This result agrees with lemma 3.3, since every schedulable P-FRP task set has to be first schedulable in the preemptive model. At best, the number of processors required in P-FRP and the preemptive would be the same.

In the optimal partition for the preemptive model, all *n* tasks required less number of processors than *n*, to be feasibly scheduled. In optimal partitioning for P-FRP, task sets requiring same number of processors as tasks in 6-task and 8-task sets is 28 and 3, respectively. All 10-task sets required less than 10 processors. Hence, in P-FRP processor requirement could be as high as the number of tasks, even though we have bounded the utilization factors of individual tasks by 0.3.

X. Related Work

Previous work on P-FRP by Kaiabachev et al [24] and Ras and Cheng [34] have provided basic schedulability conditions in a uniprocessor system. Response time analysis under symmetric multiprocessing for P-FRP has been studied by Cheng and Ras [12].

Partitioning of tasks in multiprocessor systems for the preemptive model was first studied by Dhall and Liu [16], and then improved upon by Davari and Dhall [13],[14] and Oh and Son [30]. Methods for online scheduling of tasks on multi-processor scheduling have been presented by Dertouzos and Mok [15]. Oh and Baker [29] and Lopez et al [27] provide utilization bounds that can be used to determine the schedulability of a task set for a given number of processors. Baruah and Goossens [4] also present a sufficient multiprocessor schedulability test under RM-scheduling. However, this sufficient schedulability test cannot be used in P-FRP because as per lemma 4.4, only necessary conditions of the preemptive model are applicable for P-FRP. Andersson et al [1] provide algorithms for global scheduling under fixed priority assignment.

Baruah et al [5] have studied scheduling on multiple resources using proportionally fair (*Pfair*) [6] strategy. In Pfair scheduling, execution of a task is divided into small blocks, and blocks of different tasks are executed consecutively. This is found to give a feasible schedule for multi-processor systems with low computational overhead [5]. However Pfair scheduling cannot be applied to the P-FRP execution model since the execution of a function is an 'all or nothing' proposition, and cannot be divided into computational blocks. Anderson et al [3] have presented a way to implement hard-real time transactions on multi-processors. This work does not address partitioning of tasks but changes the mechanism of implementing transactions.

Response-time analysis under multi-processing scheduling for similar execution models have been done by several authors. Holman and Anderson [21] use the Pfair technique for scheduling lock-free [3] transactions on multiple processor. Lock-free [2] is a way to access shared resources among tasks, such that none of them have to halt and wait for the resource to become available. It is a mechanism to avoid priority inversion [35] between tasks sharing resources. Comparisons between transaction memory based systems and lock-free processing and benefits of the former have been shown in Herlihy and Moss [21]. Fahmy et al [19] have presented response time analysis for transactional memory [21] under dynamic scheduling policies. Static partitioning of tasks in these execution models, as determined in our paper, have not been presented yet.

XI. Conclusion and Future Work

We have presented a new exact schedulability test algorithm for P-FRP and have used in three first-fit based partitioning algorithms. A computational method to derive an optimal partitioning scheme has also been presented. Experimental results using synthetic task sets of different sizes show that by applying the exact schedulability test presented in this paper, the existing first-fit partitioning schemes can be used for scheduling P-FRP tasks in multi-processor systems. Results also show that for P-FRP, the new first-fit partitioning scheme arranged by processing time introduced by us, performs closest to the optimal partition. The number of processors required to schedule P-FRP tasks are also higher, than the number of processors required to schedule the same tasks in a preemptive execution model.

Unlike previous work in [16],[13],[14],[30], where theoretical proofs to validate the performance of the first-fit algorithms are derived, we have used experimental tasks sets for the same. A theoretical validation for P-FRP is difficult to derive due to its dynamic nature of execution, where the actual processor time taken by a task to complete processing can me more than its defined processing time, which is known *a priori*. As part of ongoing work, we are continuing our attempts to derive theoretical proofs regarding the performance of the first-fit algorithms presented in this paper.

The first-fit partitioning algorithm presented in this paper runs in polynomial time. However, the exact schedulability test for P-FRP takes significant computation time, relative to schedulability tests for the preemptive model. Unfortunately, till the time of submission of this paper, no faster method has been developed to compute schedulability of a P-FRP task set in an inexpensive way. Development of such a method will be useful in speeding up the computation time of the first-fit algorithms and is scope for future work.

Static partitioning of tasks has a low overhead, since the partitioning algorithm has to be run only once. The use of global partitioning algorithms and ascertaining their effectiveness in the P-FRP execution model, will be an important contribution. Study under global partitioning is also important since Leung and Whitehead [25] have shown that the partitioned and global approaches for fixed priority scheduling in the preemptive model are incomparable, and there can be task sets which are schedulable by only one of these approaches. Hence, in the preemptive model, there is no single scheduling approach that is guaranteed to feasibly schedule task sets in multi-processor systems using fixed priority scheduling. Future research can determine if this statement also holds true for P-FRP. Deriving the partitioning when tasks are not offset-free, is also an important contribution towards application of P-FRP in multi-processor environments.

XII. Hardware Implementation

In ongoing work we will be testing the partitioning algorithms for P-FRP in a P8X32 Propeller [31] based multi-core platform. A similar platform has been used for analysis in [12]. The P8X32 Propeller has 8 cores (termed cogs) sharing common resources through a central hub, and the developer has full control over the usage of each cog. A shared system clock keeps all processors in the same time reference. Hence, the P8X32 Propeller has all features required by a SMP platform, is designed for use in embedded systems and is commercially used in robots and process control devices. A USB interface allows programming the chip through a PC, and using assembly instructions, an abort-restart model of execution can be implemented, to validate the effectiveness of static partitioning algorithms presented in this paper.

References

[1] B. Andersson, S. Baruah, J. Jonsson. "Static-Priority Scheduling on Multiprocessors". *RTSS'01, pp. 193-202*, 2001

[2] 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

[3] J. Anderson, R. Jain, S. Ramamurthy. " Implementing Hard Real-Time Transactions on Multiprocessors ". *Real-Time Database and Information Systems: Research Advances, Azer Bestavros and Victor Fay-Wolfe (eds.), Kluwer Academic Publishers, Norwell, MA, pp. 247-260*, 1997

[4] S. K. Baruah , J.Goossens. "Rate-Monotonic Scheduling on Uniform Multiprocessors". *IEEE Trans. Computing 52, 7, pp. 966-970*, 2003

[5] S. K. Baruah, J. Gehrke, C. G. Plaxton. "Fast scheduling of periodic tasks on multiple resources". *9th international Symposium on Parallel Processing, pp. 280-288*, 1995

[6] S.K.Baruah, N.K. Cohen, C.G. Plaxton, D.A.Varvel. "Proportionate progress: a notion of fairness in resource allocation". *ACM Symposium on Theory of Computing, pp.* 345-354, 1993

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

[8] C. Belwal , A.M.K. Cheng. "On Determining Actual Response Time in P-FRP", *Practical Aspects of Declarative Languages(PADL)'11* , 2011

[9] C. Belwal , A.M.K. Cheng. "On the Feasibility Interval for P-FRP". *Manuscript under review, http://www2.cs.uh.edu/~cbelwal/ FeasibilityInterval_PFRP .pdf*, 2010

[10] C. Belwal , A.M.K. Cheng. "On Priority Assignment in P-FRP". *RTAS'10 Work-in-Progress Session* , 2010

[11] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, S. Baruah. "A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms ". *Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Joseph Y. Leung (ed.), Chapman and Hall/CRC, Boca Raton, Florida, pages 30-1 - 30-19*, 2004

[12] Albert M. K. Cheng , Jim Ras. ``Response Time Analysis of the Abort-and-Restart Model under Symmetric Multiprocessing". *ICESS-2010*, 2010

[13] S.Davari , S.K. Dhall. "An On-Line Algorithm for Real-Time Tasks Allocation". *RTSS '86*, 1986

[14] S.Davari , S.K. Dhall. "On a Real-time Task Allocation Problem". *19th annual Hawaii International Conf. on System Sciences*, 1985

[15] M.L. Dertouzos , A.K. Mok. "Multiprocessor Online Scheduling of Hard-Real-Time Tasks". *IEEE Trans. Soft. Eng. 15, 12, pp.1497-1506*, 1989

[16] S.K. Dhall , C.L. Liu. "On a real-time scheduling problem". *Operation Research, 26(1):127-140*, 1978

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

[18] Erlang, *http://www.erlang.org*

[19] S. F. Fahmy, B. Ravidran, E. Jensen. "On bounding response times under software transactional memory in distributed multiprocessor real-time systems". *DATE'09*, 2009

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

[21] 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

[22] P. Holman , J. H. Anderson. “Supporting lock-free synchronization in Pfair-scheduled real-time systems”. *Journal of Parallel Distrib. Comput. 66 (1), pp. 47-67*, 2006

[23] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey ,R.L. Graham. “Worst case performance bounds for simple One-dimensional packing Algorithms”. *SIAM Journal of Computing, Vol. 3, pp. 299-325*, 1974

[24] R. Kaiabachev, W. Taha , A. Zhu. “E-FRP with Priorities”. *EMSOFT’07 , pp. 221-230* , 2007

[25] J.Y.T. Leung, J. Whitehead. “On the complexity of fixed-priority scheduling of periodic, real-time tasks”, *Performance Evaluation (Netherlands) 2(4), pp. 237-250*, 1982

[26] 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

[27] J.M. Lopez, J.L. Diaz, D.F. Garcia. “Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling”. *IEEE Trans. Parallel Distrib. Syst. 15, 7, pp. 642-653*, 2004

[28] A.K. Mok. “Fundamental Design Problems of Distributed Systems For the Hard-real-time Environment”. *Technical Report. UMI Order Number: TR-297, Massachusetts Institute of Technology*, 1983

[29] D. Oh, T.P. Baker. “Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment”. *Real-Time Syst. 15(2), pp.183-192*, 1998

[30] Y. Oh, S. Son. “Fixed-Priority Scheduling of Periodic Tasks on Multiprocessor Systems”. *Technical Report. UMI Order Number: CS-95-16., University of Virginia*. 1995

[31] Parallax P8X32 Properller Chip: http://www.parallax.com

[32] J. Peterson, G. D. Hager, P. Hudak. “A Language for Declarative Robotic Programming”. *ICRA’99* , 1999

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

[34] J. Ras, A.M.K. Cheng. “Response Time Analysis for the Abort-and-Restart Task Handlers of the Priority-Based Functional Reactive Programming (P-FRP) Paradigm”, *RTCSA’09*, 2009

[35] 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

[36] M. Swaine, “It's Time to Get Good at Functional Programming”. *Dr. Dobbs Journal, http://www.drdobbs.com, Dec ‘08*, 2008

[37] Z. Wan, W. Taha, P. Hudak. “Real - time FRP”. *ICFP’01, pp. 146-156*, ACM Press ,2001

[38] Z. Wan, W. Taha, P. Hudak. “Task driven FRP”. *PADL’02, Lecture Notes on Computer Science*, Springer, 2002

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