CHALMERS | GÖTEBORG UNIVERSITY

Technical Report No. 2009-7

Load Regulating Algorithm for Static-Priority Task Scheduling on Multiprocessors

RISAT MAHMUD PATHAN
JAN JONSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY/
UNIVERSITY OF GOTHENBURG

Load Regulating Algorithm for Static-Priority Task Scheduling on Multiprocessors

Risat Mahmud Pathan and Jan Jonsson
Technical Report 2009-7
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96, Göteborg, Sweden
{*risat, janjo*}*@chalmers.se*

Abstract

This paper proposes a fixed-priority partitioned scheduling algorithm for periodic tasks on multiprocessors. A new technique for assigning tasks to processors is developed and the schedulability of the algorithm is analyzed for worst-case performance. We prove that, if the workload (utilization) of a given task set is less than or equal to 55.2% of the total processing capacity on m processors, then all tasks meet their deadlines. During task assignment, the total work load is regulated to the processors in such a way that a subset of the processors are guaranteed to have an individual processor load of at least 55.2%. Due to such load regulation, our algorithm can be used efficiently as an admission controller for online task scheduling. And this online algorithm is scalable with increasing number of cores in chip multiprocessor.

In addition, our scheduling algorithm possesses two properties that may be important for the system designer. The first one guarantees that if task priorities are fixed before task assignment they do not change during task assignment and execution, thereby facilitating debugging during development and maintenance of the system. The second property guarantees that at most $m/2$ tasks are split, thereby keeping the run-time overhead as caused by task splitting low.

1 Introduction

In recent years, applications of many embedded systems run on multiprocessors, in particular, chip multiprocessors [1, 2]. The main reasons for doing this is to reduce power consumption and heat generation. Many of these embedded systems are also hard real-time systems in nature and meeting the task deadlines of the application is a major challenge. Since many well-known uniprocessor scheduling algorithms, like Rate-Monotonic (RM) or Earliest Deadline First (EDF) [3], are no longer optimal for multiprocessors [4], developing new scheduling algorithms for multiprocessor platform have received considerable attention. In this paper, we consider systems where each periodic task has its deadline equal to its period. We address the problem of meeting deadlines for a set of n periodic tasks using preemptive static-priority scheduling on m processors. We also propose an extension of our scheduling algorithm that can be efficiently used as an admission controller for online scheduling.

Static-priority preemptive task scheduling on multiprocessor can be classified as *global* or *partitioned* scheduling. In global scheduling, at any time m highest-priority tasks from a global queue are scheduled on m processors. In partitioned scheduling, a task is allowed to execute only on one fixed, assigned, processor. In global scheduling, tasks are allowed to migrate while

in partitioned scheduling, tasks are never allowed to migrate. Many static-priority scheduling policies for both global [5, 6, 7, 8] and partitioned [4, 9, 10, 11, 12, 13, 14, 15] approaches have been well studied. In this paper, a variation of partitioned scheduling technique in which a bounded number of tasks can migrate to a different processor is addressed.

It has already been proved that, there exists some task set with load slightly greater than 50% of the capacity of a multiprocessor platform on which a deadline miss must occur for both global and partitioned static-priority scheduling [5, 15]. To achieve higher utilization bound than 50%, some recent work proposes techniques where *migratory* [16] or *split* [17, 18, 19] tasks are allowed to migrate using a variation of partitioned scheduling for dynamic-priority tasks. Very little work [20] have addressed the scheduling problem for static-priority tasks using task splitting to overcome the 50% utilization bound. We propose a static-priority scheduling algorithm, called *Interval Based Partitioned Scheduling* (`IBPS`), for periodic tasks using task splitting approach. By *task splitting*, we mean that some tasks are allowed to migrate their execution to a different processor during execution[1]. We call the task that is splitted a 'split task' and its pieces 'subtasks'. No single task or the subtasks of a split task can run in parallel. In `IBPS`, rate-monotonic (RM) prioritization [3] is used both during task assignment and during run-time scheduling of tasks on a processor. One of the main contributions of this paper is to prove that *if the total utilization (or workload) of a set of n periodic tasks is less than or equal to 55.2% of the capacity of m processors, the task set is RM schedulable on m processors using `IBPS`.*

Apart from the guarantee bound, the important features of `IBPS` are:

- During task assignment, the individual processor loads are regulated in a way that makes on-line scheduling (task addition and removal) more efficient than for other existing task-splitting algorithms. Due to load regulation, a bounded number of processors have load less than 55.2%. So, the percentage of processors with load greater than 55.2% increases as the number of processors in a system increases. Therefore, with increasing number of cores in chip multiprocessor (e.g. Sun's Rock processor with 16 cores [2]), our proposed online scheduling is more effective and scales very well.

- The priority of a task given before task assignment is not changed to another priority during task assignment and execution, which facilitates debugging during system development and maintenance.

- The task splitting algorithm split tasks such that the number of migrations is lower than for other existing task-splitting algorithms.

The rest of the paper is organized as follows. In Section 2, the important features of `IBPS` are further elaborated. In Section 3, we present the assumed system model. In Section 4, we briefly discuss the basic idea of our task assignment algorithms and also present our task splitting approach. Then, in Sections 5 through 7, we present the `IBPS` task-assignment algorithms in detail. The performance of `IBPS` and its online version is presented in Section 8 and 9, respectively. In Section 10, we discuss other work related to ours. Section 11 concludes the paper.

2 Important Features of `IBPS`

A real-time task scheduling algorithm does not only require worst-case guarantee to meet deadlines but also need to be practically implementable. In addition to the utilization bound of more than 50%, the algorithm `IBPS` has three other major features: (i) load regulation, (ii) priority traceability property, and (iii) low cost of splitting.

[1]Here, we do not mean splitting the code.

Load Regulation: IBPS regulates the load of a processor during task assignment. When assigning tasks to processors, the objective of IBPS is to have as many processors as possible with load greater than 55.2% and still meet all deadlines. In the worst-case, the number of processors on which the load is less than or equal to 55.2% is at most $min\{m,4\}$. This way of load regulation bound the number of underutilized processor.

Load regulation of IBPS enables design of efficient admission controller for online scheduling. In practice, many real-time systems are dynamic in nature, that is, tasks arrive and leave the system online. After accepting an online task, we then need to assign the task to a particular processor. Finding the best processor to assign the task may require disturbing the existing schedule in all m processors by means of a reassignment of tasks (e.g. task assignment algorithms that require sorting).

As will be evident later when IBPS is used online, finding the best processor to which an accepted online task is assigned requires searching at most $min\{m,4\}$ processors (the under-loaded processors). Similarly, when a task leaves the system, reassignment of tasks undergo in at most $min\{m,5\}$ processors to regulate the load for future admittance of new tasks. IBPS runs in linear time, therefore, reassignment of tasks on a bounded number of processors for load regulation is efficient. Moreover, task reassignment on a bounded number of processors makes our online scheduling algorithm scalable with the trend of increasing number of cores in chip multiprocessor.

Priority Traceability Property: From a system designer's point of view it is desirable to facilitate debugging (execution tracing), during the development and maintenance of the system. One explanation for the wide-spread use of RM in industry is the relative ease with which the designer can predict the execution behaviour at run-time. The dynamic-priority EDF scheduler has (despite being just as mature a scheduling method and having stronger schedulability properties) not received a corresponding attention in industrial applications. Even for static-priority schedulers, the ease of debugging differs for different algorithms. For example, when studying the recent work in [20] of the static-priority partitioned scheduling with task splitting, we see that it is possible that the deadline of a subtask could become smaller than the given deadline of the original task during task assignment to processor. This in turn, could make the priorities of the subtasks different from the original task and therefore cause a different, less traceable, execution behavior. A similar behavior can be identified in known dynamic-priority task-splitting algorithms [17, 21, 18, 22, 19] where subtask of split task may have different priority (i.e. executes in specific time slots, or has smaller deadline). IBPS is a static-priority partitioned algorithm with a strong *priority traceability property*, in the sense that if the priorities of all tasks are fixed before task assignment they never change during task assignment and execution.

Cost of Splitting: One final property of interest for task-splitting partitioned scheduling in particular is the run-time penalty introduced due to migration. Clearly, the number of total split tasks and number of subtasks (resulting from a split tasks) directly affect the amount of preemptions and migrations, both of which in turn may affect cache performance and other migration related overhead. Therefore, it is always desirable to reduce the number of split tasks and reduce the number of subtask for each split task. For all existing dynamic- and static-priority task-splitting algorithms, the number of split tasks is $(m-1)$ on m processor. In IBPS, total number of split task in the worst-case is at most $m/2$. In IBPS, a split task has only two subtasks and therefore, a split task never suffers more than once due to migration in one period.

3 System Model

In this work, we assume a task set Γ consisting of n periodic tasks. Each task $\tau_i \in \Gamma$ arrives repeatedly with a period T_i and requires C_i units of worst-case execution time within each period. Each task has an implicit deadline equal to its period, and task priorities are assigned according to the RM policy (lower the period, the higher the priority). We define the *utilization* of task

τ_i as $u_i = \frac{C_i}{T_i}$. The *load* or *total utilization* of any task set A is $U(A) = \sum_{\tau_i \in A} u_i$. When a task τ_i is split, it is considered as two subtasks, τ_i' and τ_i'', such that both subtasks has execution time and period equal to $\frac{C_i}{2}$ and T_i, respectively. Note that, since the period of a subtask is equal to the period of the split task τ_i, we must have $u_{i'} = u_{i''} = \frac{u_i}{2}$. When assigning tasks to each processor, we use Liu and Layland's sufficient feasibility condition for RM scheduling in [3] for determining whether the tasks can be assigned to a processor. According to [3], the Liu and Layland's test is, *if* $U(A) \leq n(2^{\frac{1}{n}} - 1)$, *all* n *tasks in set* A *meet deadlines in one processor.* Here, we make the pessimistic assumption that each non-split task has an offset $\phi_i = 0$. However, the second subtask τ_i'' of a split task τ_i is given an offset equal to $\phi_{i''} = \frac{C_i}{2}$ to ensure nonparallel execution with first subtask τ_i'. In rest of the paper, we use the notation $\text{LLB}(n)=n(2^{\frac{1}{n}} - 1)$ for n tasks, $\text{LLB}(\infty)=\ln 2$ to represent an unknown number of tasks, and also let $Q=(\sqrt{2}\text{-}1)$.

4 Task Assignment and Splitting Overview

Our proposed task assignment algorithm starts by dividing the utilization interval (0,1] into seven disjoint utilization subintervals I_1–I_7 (see Table 1). For $a < b$, we use I_a–I_b to denote all

$$\text{I}_1 = (\tfrac{4Q}{3}, 1] \quad \text{I}_2 = (\tfrac{8Q}{9}, \tfrac{4Q}{3}] \quad \text{I}_3 = (\tfrac{2Q}{3}, \tfrac{8Q}{9}]$$

$$\text{I}_4 = (\tfrac{8Q}{15}, \tfrac{2Q}{3}] \quad \text{I}_5 = (\tfrac{4Q}{9}, \tfrac{8Q}{15}] \quad \text{I}_6 = (\tfrac{Q}{3}, \tfrac{4Q}{9}]$$

$$\text{I}_7 = (0, \tfrac{Q}{3}] \qquad (\textit{where, } Q = \sqrt{2} - 1)$$

Table 1: Seven disjoint utilization subintervals I_1–I_7

the subintervals I_a, I_{a+1}, ... I_b. Note that, each task $\tau_i \in \Gamma$ will have a utilization that belongs to exactly one of the subintervals I_1–I_7. By overloading the set membership operator "$\in$", we write $\tau_i \in \text{I}_k$ to denote "τ_i in I_k=(a,b]" for any $k \in \{1 \ldots 7\}$, if $a < u_i \leq b$. For example, if a task τ_i has $u_i = \frac{4Q}{5}$, then $\tau_i \in \text{I}_3$. Clearly, the grouping of tasks into subintervals can be completed in linear time.

Why Seven Utilization Subintervals? The seven utilization intervals result from our four different strategies for task assignment: (i) low number of subtasks per split task, (ii) low number of split tasks, (iii) assigning low number of tasks per processor[2], and (iv) load regulation.

Following these four task assignment strategies, we start by assigning only one task to one processor exclusively. To avoid assigning a task with very small utilization to one processor exclusively, we need to select a task that belongs to certain utilization subintervals. If `IBPS` has worst-case utilization bound U_w and load regulation try to maintain load on most processors beyond U_w, then one task with utilization greater than U_w is assigned to one processor exclusively. Thus, we obtain our first utilization interval which is $(\text{U}_w, 1]$. The exact value of U_w is determined when we assign more than one task having utilization less than U_w to one processor. When we try to assign two tasks with utilization less than U_w to one processor, we find that if these two tasks have equal utilization, then each task's utilization can not be greater than $\text{Q}=(\sqrt{2}\text{–}1)$ according to $\text{LLB}(n{=}2)$. This implies $\text{U}_w \leq 41\%$ without task splitting technique. To achieve U_w greater than 50%, we split a task with utilization less than U_w in two subtasks each having same utilization. We gain no advantage by having unequal utilization for the two subtasks as individual task utilization is bounded from below and our strategy is to assign minimum number of tasks per processor. So, each subtask has utilization at most $\frac{\text{U}_w}{2}$. We assign one such subtask and a non-split task to one processor. For RM schedulability, we must have $\text{U}_w + \frac{\text{U}_w}{2} \leq 2Q$. This

[2]According to LLB, the RM scheduling on uniprocessor achieves higher utilization bound if number of tasks assigned to the processor is small [3].

implies the value of $U_w \leq \frac{4Q}{3}$ and we get our first utilization subinterval I_1=($\frac{4Q}{3}$,1]. Thus, this interval also defines the maximum possible utilization bound of IBPS.

The overall goal with the task assignment is to achieve the maximum possible U_w, which is $\frac{4Q}{3}$ ≈55.2%. To achieve this bound, at each stage of task assignment, we bound the utilization of tasks from below for load regulation by selecting a task from certain utilization interval, split a task in only two subtasks when task splitting is unavoidable and try to assign minimum number of tasks to one processor. The consequence is that, we derive the seven utilization intervals I_1-I_7 given in Table 1. The derivation of I_2-I_6 are done in a similar fashion as for I_1, while I_7 becomes the remaining interval task utilization in which requires no lower bound. More details regarding the derivation are available in Appendix A.

Task Assignment Overview: IBPS assigns tasks to processors in *three* phases. In the *first phase*, tasks from each subinterval I_k are assigned to processors using one particular *policy* for each I_k. Any unassigned tasks in I_2–I_6 after first phase, called *odd tasks*, are assigned to processors in the *second phase*. If tasks are assigned to m' processors during the first and second phases, the total utilization in each of the m' processors will then be greater than $\frac{4Q}{3} \approx 55.2\%$ due to load regulation strategy. Any unassigned tasks after the second phase, called *residue tasks*, are assigned to processors during the *third phase*. If, after the second phase, the total utilization of the residue tasks is smaller than or equal to $\frac{4Qm''}{3}$ for the smallest non-negative integer m'', all the residue tasks are assigned to at most m'' processors in third phase. The load on these m'' processors may be smaller than 55.2%. Load regulation in first two phases ensures that $m'' \leq 4$ in third phase. When task arrives online, we need only to consider these m'' processor for task assignment. The different task assignment algorithms in the three phases constitute the algorithm IBPS.

We use $U_{\text{MN}}(A)$ and $U_{\text{MX}}(A)$ to denote the lower and upper bound, respectively, on the total utilization of all tasks in an arbitrary task set A:

$$U_{\text{MN}}(A) = \sum_{\tau_i \in A} x \quad \text{where } \tau_i \in I_k = (x, y] \tag{1}$$
$$U_{\text{MX}}(A) = \sum_{\tau_i \in A} y \quad \text{where } \tau_i \in I_k = (x, y]$$

Clearly, the following inequality holds for any task set A:

$$U_{\text{MN}}(A) < U(A) \leq U_{\text{MX}}(A) \tag{2}$$

If IBPS assigns a task set to m_x processors and $m_x \leq m$, we declare SUCCESS for schedulability on m processors. Otherwise, we declare FAILURE. Therefore, IBPS can be used to determine: (i) the number of processors needed to schedule a task set, and (ii) whether a task set is schedulable on m processors.

Task Splitting Algorithm: IBPS uses the algorithm SPLIT in Fig. 1 for task splitting. The input to algorithm SPLIT is a set X, containing an odd number of tasks. Each task in X, except the highest-priority one, is assigned to one of two selected (empty) processors. The highest-priority task is split and assigned to both processors. As will be evident in later sections, no more tasks will be assigned to these processors.

The highest priority task $\tau_k \in X$ is determined (line 1) and τ_k is split into two subtasks τ_k' and τ_k'' such that $u_{k'} = u_{k''} = \frac{u_k}{2}$ (line 2-4). Half of the tasks from set $(X - \{\tau_k\})$ is stored in a new set X_1 (line 5), and the remaining half in another new set X_2 (line 6). Note that $|X - \{\tau_k\}|$ is an even number. Subtask $\tau_k{}'$ and all tasks in X_1 are assigned to one processor (line 7), while subtask $\tau_k{}''$ and all tasks in X_2 are assigned to the other processor (line 8). Note that, since τ_k is the highest-priority task and no more tasks will be assigned to the selected processors, the offset of $\tau_k{}''$ ($\phi_{k''}$=$\frac{C_k}{2}$) will ensure that the two subtasks are not executed in parallel.

Algorithm SPLIT (TaskSet: X)
1. Let $\tau_k \in X$ such that $T_k \leq T_i$ for all $\tau_i \in X$
2. Split the task τ_k into subtasks τ_k' and τ_k'' such that
3. $\phi_k{}'$=0, $C_k' = C_k/2$, $T_k' = T_k$
4. $\phi_k{}''$=$C_k/2$, $C_k'' = C_k/2$, $T_k'' = T_k$
5. Let task set X_1 contain $\frac{|X-\{\tau_k\}|}{2}$ tasks from $X - \{\tau_k\}$
6. $X_2 = X - X_1 - \{\tau_k\}$
7. Assign $\tau_k{}'$ and all tasks in set X_1 to one processor
8. Assign $\tau_k{}''$ and all tasks in set X_2 to one processor

Figure 1: Task Splitting Algorithm

5 Task Assignment: First Phase

During the first phase, IBPS assigns tasks using a particular policy for each of the seven subintervals using load regulation strategy. Common for all policies, however, is that each processor to which tasks have been assigned will have an RM schedulable task set with a total utilization strictly greater than $\frac{4Q}{3}$ for load regulation. We now describe the seven policies used during the first task-assignment phase.

Policy 1: Each task $\tau_i \in \mathrm{I}_1$=$(\frac{4Q}{3}, 1\}$ is assigned to one dedicated processor. Clearly, each of these tasks is trivially RM schedulable with a utilization greater than $\frac{4Q}{3}$ on one processor. This policy guarantees that there will be no tasks left in I_1 after the first phase.

Policy 2: Exactly *three* tasks in I_2=$(\frac{8Q}{9}, \frac{4Q}{3}]$ are assigned to *two* processors using algorithm SPLIT given in Fig. 1. This process iterates until less than three tasks are left in I_2. Thus, there are 0–2 unassigned tasks in I_2 to declare as *odd* tasks.

The proof that the tasks assigned to each processor are RM schedulable and the utilization on each processor is greater than $\frac{4Q}{3}$ is as follows. Assume that a particular iteration assigns the three tasks τ_k, τ_i and τ_j from I_2 to two processors by calling SPLIT ($\{\tau_k$,τ_i,$\tau_j\}$) such that τ_k is the highest priority tasks. Then, line 3 of algorithm SPLIT ensures that $\frac{4Q}{9} < u_{k'} \leq \frac{2Q}{3}$. Now, without loss of generality, assume $X_1 = \{\tau_i\}$ in line 5 of SPLIT. Using Eq.(1), we have $U_{\mathrm{MN}}(\{\tau_i, \tau_k'\}) = \frac{8Q}{9} + \frac{4Q}{9} = \frac{4Q}{3}$ and $U_{\mathrm{MX}}(\{\tau_i, \tau_k'\}) = \frac{4Q}{3} + \frac{2Q}{3} = 2Q = LLB(2)$. Using Eq. (2) we have, $\frac{4Q}{3} < U(\{\tau_i, \tau_{k'}\}) \leq LLB(2)$. Similarly we have, $\frac{4Q}{3} < U(\{\tau_j, \tau_{k''}\}) \leq LLB(2)$.

Policy 3: Exactly *two* tasks from I_3=$(\frac{2Q}{3}, \frac{8Q}{9}]$ are assigned to one processor without any splitting. This process iterates until less than two tasks are left in I_3. Thus, there are 0–1 task left in I_3 to declare as *odd* tasks. Two tasks in I_3 must have a total utilization greater than $\frac{2\times 2Q}{3}$ and less than or equal to $\frac{2\times 8Q}{9}$, assigned to one processor. Since $\frac{16Q}{9} < LLB(2)=2Q$, the two tasks from I_3 are RM schedulable with processor utilization greater than $\frac{4Q}{3}$.

Policy 4: Exactly *five* tasks from I_4=$(\frac{8Q}{15}, \frac{2Q}{3}]$ are assigned to *two* processors using algorithm SPLIT in Fig. 1. This process iterates until there are less than five tasks left in I_4. Thus, there are 0–4 unassigned tasks in I_4 to declare as *odd* tasks.

The proof that the tasks in each processor are RM schedulable and the utilization in each processor is greater than $\frac{4Q}{3}$ is as follows. Assume that a particular iteration assigns the five tasks τ_k, τ_i, τ_j, τ_a and τ_b from I_4 to two processors by calling SPLIT ($\{\tau_k$,τ_i,τ_j,τ_a,$\tau_b\}$) such that τ_k is the highest priority tasks. Then, line 3 of algorithm SPLIT ensures that $\frac{4Q}{15} < u_{k'} \leq \frac{Q}{3}$. Now, without loss of generality, assume $X_1 = \{\tau_i, \tau_j\}$ in line 5 of SPLIT . Using Eq.(1), we have $U_{\mathrm{MN}}(\{\tau_i, \tau_j, \tau_k'\}) = \frac{8Q}{15} + \frac{8Q}{15} + \frac{4Q}{15} = \frac{4Q}{3}$ and $U_{\mathrm{MX}}(\{\tau_i, \tau_j, \tau_k'\}) = \frac{2Q}{3} + \frac{2Q}{3} + \frac{Q}{3} = \frac{5Q}{3} < LLB(3)$. Using Eq. (2) we have, $\frac{4Q}{3} < U(\{\tau_i, \tau_j, \tau_{k'}\}) \leq LLB(3)$. Similarly we have, $\frac{4Q}{3} < U(\{\tau_a, \tau_b, \tau_{k''}\}) \leq LLB(3)$.

Policy 5 : Exactly *three* tasks from I_5=$(\frac{4Q}{9}, \frac{8Q}{15}]$ are assigned to one processor. This process iterates until there are less than three tasks left in I_5. Thus, there are 0–2 tasks left in I_5 to declare as *odd* tasks. Three tasks in I_5 must have a total utilization greater than $\frac{3\times 4Q}{9}$ and less

than $\frac{3\times 8Q}{15} < LLB(3)$. So, each processor utilization is greater than $\frac{4Q}{3}$ and the three tasks in I_5 are RM schedulable on one processor.

Policy 6: Exactly four tasks from $I_6=(\frac{Q}{3}, \frac{4Q}{9}]$ are assigned to one processor. This process iterates until there is less than four tasks left in I_6. Thus, there are 0–3 tasks left in I_6 to declare as *odd* tasks. Four tasks in I_6 must have a utilization greater than $\frac{4\times Q}{3}$ and less than $\frac{4\times 4Q}{9} < LLB(4)$. So, each processor utilization is greater than $\frac{4Q}{3}$ and the four tasks in I_6 are RM schedulable on one processor.

Policy 7: In this policy, `IBPS` assigns tasks from $I_7=\{0, \frac{Q}{3}]$ using First-Fit (FF) bin packing allocation as in [4]. We denote the l^{th} processor by Θ_l and the total utilization (load) of Θ_l by $L(\Theta_l)$. Assume that Θ_p is the first considered processor and that Θ_q is the last considered processor, for $q \geq p$ using FF, in this policy. When a task $\tau_i \in I_7$ cannot be feasibly assigned to Θ_l, for $l = p, (p+1), \ldots (q-1)$ we must have $L(\Theta_l) + u_i > LLB(\infty) = \ln 2$. Since $u_i \leq \frac{Q}{3}$, we have $L(\Theta_l) > \ln 2 - \frac{Q}{3} > \frac{4Q}{3}$. So, the total utilization of the tasks in each processor is greater than $\frac{4Q}{3}$, except possibly the last processor Θ_q. If $L(\Theta_q) \leq \frac{4Q}{3}$, the task assignment to Θ_q is undone and these unassigned tasks are called *residue* tasks in I_7. If $L(\Theta_q) > \frac{4Q}{3}$, all the tasks in I_7 assigned to processors $\Theta_p, \Theta_{p+1}, \ldots \Theta_q$ are RM schedulable with utilization in each processor greater than $\frac{4Q}{3}$.

In rest of the paper, to denote the set of residue tasks in I_7 we use $S = \{\tau_r | \tau_r$ is a residue task in $I_7\}$. We have the following Lemma 1.

Lemma 1. *All the residue task in S are RM schedulable in one processor.*

Proof. From policy 7, we have $U(S) \leq \frac{4Q}{3}$ (undone task assignment in Θ_q). Since $\frac{4Q}{3} < LLB(\infty) = \ln 2$, all tasks in S are RM schedulable in one processor. □

Note that, first phase of task assignment runs in linear time due to its iterative nature.

6 Task Assignment: Second Phase

During the second phase, `IBPS` assigns unassigned tasks in subintervals I_2–I_6, referred to as *odd tasks*, using algorithm OddAssign in Fig. 2. Unlike the first phase, however, each processor can now be assigned tasks from more than one subinterval. Using algorithm OddAssign, tasks assigned in each iteration of each `while` loop are RM schedulable with total utilization is greater than $\frac{4Q}{3}$ on each processor due to load regulation strategy. We prove this by showing that the inequality $\frac{4Q}{3} < U(A) \leq LLB(|A|)$ holds (where A is the task set assigned to one processor) in each iteration of each `while` loop (here, named Loop 1–Loop 6). We now analyze each loop separately.

Loop 1 (line 1–2): Each iteration of this loop assigns A=$\{\tau_i,\tau_j\}$ to one processor such that $\tau_i \in I_2 = (\frac{8Q}{9}, \frac{4Q}{3}]$ and $\tau_j \in I_4 = (\frac{8Q}{15}, \frac{2Q}{3}]$. Thus, $U_{\text{MX}}(\text{A}) = \frac{4Q}{3} + \frac{2Q}{3} = 2Q = LLB(2)$ and $U_{\text{MN}}(\text{A}) = \frac{8Q}{9} + \frac{8Q}{15} = \frac{64Q}{45} > \frac{4Q}{3}$.

Loop 2 (line 3–4): Each iteration of this loop assigns A=$\{\tau_i,\tau_j\}$ to one processor such that $\tau_i \in I_2 = (\frac{8Q}{9}, \frac{4Q}{3}]$ and $\tau_j \in I_5 = (\frac{4Q}{9}, \frac{8Q}{15}]$. Thus, $U_{\text{MX}}(A) = \frac{4Q}{3} + \frac{8Q}{15} = \frac{28Q}{15} < LLB(2)$ and $U_{\text{MN}}(A) = \frac{8Q}{9} + \frac{4Q}{9} = \frac{4Q}{3}$.

Loop 3 (line 5–6): Each iteration of this loop assigns A=$\{\tau_i,\tau_j,\tau_k\}$ to one processor such that $\tau_i \in I_3 = (\frac{2Q}{3}, \frac{8Q}{9}]$, $\tau_j \in I_6 = (\frac{Q}{3}, \frac{4Q}{9}]$ and $\tau_k \in I_6 = (\frac{Q}{3}, \frac{4Q}{9}]$. Thus, $U_{\text{MX}}(A) = \frac{8Q}{9} + \frac{4Q}{9} + \frac{4Q}{9} = \frac{16Q}{9} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{2Q}{3} + \frac{Q}{3} + \frac{Q}{3} = \frac{4Q}{3}$.

Loop 4 (line 7–13): Each iteration of this loop assigns three tasks A=$\{\tau_i,\tau_j,\tau_k\}$ to one processor, selecting the tasks from two subintervals $I_4= (\frac{8Q}{15}, \frac{2Q}{3}]$ and $I_5=(\frac{4Q}{9}, \frac{8Q}{15}]$. Tasks are assigned to one processor either if (i) $\tau_i \in I_4$, $\tau_j \in I_5$ and $\tau_k \in I_5$ or (ii) $\tau_i \in I_4$, $\tau_j \in I_4$ and $\tau_k \in I_5$. When (i) is true, we have $U_{\text{MX}}(A) = \frac{2Q}{3} + \frac{2\times 8Q}{15} = \frac{26Q}{15} < LLB(3)$ and $U_{\text{MN}}(A) =$

Algorithm OddAssign (Odd tasks in I_2–I_6):

1. **while** both I_2 and I_4 has at least one task
2. Assign $\tau_i \in I_2$ and $\tau_j \in I_4$ to a processor
3. **while** both I_2 and I_5 has at least one task
4. Assign $\tau_i \in I_2$ and $\tau_j \in I_5$ to a processor
5 **while** I_3 has one task and I_6 has two tasks
6. Assign $\tau_i \in I_3$, $\tau_j \in I_6$ and $\tau_k \in I_6$ to a processor
7. **while** ((I_4 has one task and I_5 has two tasks)
8. **or** (I_4 has two tasks and I_5 has one task))
9. **if** (I_4 has one task and I_5 has two tasks) **then**
10. Assign $\tau_i \in I_4$, $\tau_j \in I_5$ and $\tau_k \in I_5$ to a processor
11. **else**
12. Assign $\tau_i \in I_4$, $\tau_j \in I_4$ and $\tau_k \in I_5$ to a processor
13. **end if**
14. **while** I_4 has two tasks and I_6 has one task
15. Assign $\tau_i \in I_4$, $\tau_j \in I_4$ and $\tau_k \in I_6$ to a processor
16. **while** each I_3, I_5 and I_6 has one task
17. Assign $\tau_i \in I_3$, $\tau_j \in I_5$ and $\tau_k \in I_6$ to a processor

Figure 2: Assignment of odd tasks in I_2–I_6

$\frac{8Q}{15} + \frac{2\times 4Q}{9} = \frac{64Q}{45} > \frac{4Q}{3}$. When (ii) is true, we have $U_{\text{MX}}(A) = \frac{2\times 2Q}{3} + \frac{8Q}{15} = \frac{28Q}{15} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{2\times 8Q}{15} + \frac{4Q}{9} = \frac{68Q}{45} > \frac{4Q}{3}$.

Loop 5 (line 14–15): Each iteration of this loop assigns A=$\{\tau_i,\tau_j,\tau_k\}$ to one processor such that $\tau_i \in I_4 = (\frac{8Q}{15}, \frac{2Q}{3}]$, $\tau_j \in I_4 = (\frac{8Q}{15}, \frac{2Q}{3}]$ and $\tau_k \in I_6 = (\frac{Q}{3}, \frac{4Q}{9}]$. Thus, $U_{\text{MX}}(A) = \frac{2\times 2Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{2\times 8Q}{15} + \frac{Q}{3} = \frac{7Q}{5} > \frac{4Q}{3}$.

Loop 6 (line 16–17): Each iteration of this loop assigns A=$\{\tau_i,\tau_j,\tau_k\}$ to one processor such that $\tau_i \in I_3 = (\frac{2Q}{3}, \frac{8Q}{9}]$, $\tau_j \in I_5 = (\frac{4Q}{9}, \frac{8Q}{15}]$ and $\tau_k \in I_6 = (\frac{Q}{3}, \frac{4Q}{9}]$. Thus, $U_{\text{MX}}(A) = \frac{8Q}{9} + \frac{8Q}{15} + \frac{4Q}{9} = \frac{28Q}{15} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{2Q}{3} + \frac{4Q}{9} + \frac{Q}{3} = \frac{13Q}{9} > \frac{4Q}{3}$.

Using Eq. (2), we can thus conclude that, for each iteration of each loop if task set A is assigned to one processor, we have $\frac{4Q}{3} < U(A) \leq LLB(|A|)$. The second task-assignment phase also runs in linear time due to its iterative nature.

Residue tasks: Tasks in I_2–I_6 that are still unassigned after the second phase are called *residue tasks*. For example, if there are only two unassigned tasks in I_2 after the first phase, these two odd tasks cannot be assigned to a processor in the second phase. Such a scenario, henceforth referred to as a *possibility*, of residue tasks will have to be handled during the third phase. We identify all such possibilities of residue tasks in subintervals I_2–I_6 for any task set. In particular, we determine the number of residue tasks in each of the subintervals I_2–I_6 for each identified possibility.

After the first phase, I_2 has 0–2 odd tasks, I_3 has 0–1 odd task, I_4 has 0–4 odd tasks, I_5 has 0–2 odd tasks, and I_6 has 0–3 odd tasks (see Section 5). Odd tasks thus exist, in subintervals I_2–I_6, as one of $(3 \times 2 \times 5 \times 3 \times 4 =)360$ possibilities after the first phase. During the second phase, algorithm OddAssign is able to assign *all* the odd tasks in subintervals I_2–I_6 for 316 out of the 360 possibilities of odd tasks after the first phase. It is easy to see that, this fact can be verified by running the algorithm OddAssign for each the 360 possibilities of odd tasks after first phase and by counting how many possibilities remain unassigned (please also see Appendix B for a formal proof). Therefore, for any task set, *those residue tasks in subintervals* I_2–I_6 *that need to be handled in the third phase exist as one of the remaining 44 different possibilities.* During the third phase, `IBPS` considers assigning any such possibility of residue tasks from I_2–I_6

including all residue tasks from I_7 to processors.

We now define some functions that will be used in the next sections. Function U_{RT} denotes the total utilization of all the residue tasks in I_2–I_7:

$$U_{\mathrm{RT}} = \sum_{k=2}^{7} \sum_{\tau_i \in I_k} u_i \quad [\tau_i \text{ is residue task in } I_k] \tag{3}$$

Functions U_{RMN} and U_{RMX} denote the lower and upper bound, respectively, on total utilization of all residue tasks $\tau_i \in I_k$ for $i = 2, \ldots 6$:

$$U_{\mathrm{RMN}} = \sum_{k=2}^{6} \sum_{\tau_i \in I_k} x \quad [\tau_i \in I_k{=}(x, y] \text{ is residue task}]$$

$$U_{\mathrm{RMX}} = \sum_{k=2}^{6} \sum_{\tau_i \in I_k} y \quad [\tau_i \in I_k{=}(x, y] \text{ is residue task}]$$

It is clear that, if at least one of I_2–I_6 is nonempty, we have:

$$U_{\mathrm{RMN}} < U_{\mathrm{RT}} - U(S) \leq U_{\mathrm{RMX}} \quad [S{=}\textit{ residue in } I_7] \tag{4}$$

7 Task Assignment: Third Phase

During the third phase, `IBPS` assigns all residue tasks to processors, thereby completing the task assignment. Each of the 44 *possibilities* of residue tasks in I_2–I_6 is listed in a separate row of Table 2. The columns of the table are organized as follows. The first column represents the *possibility* number. Columns two through six represent the number of residue tasks in each of the subintervals I_2–I_6, while the seventh column represents the total number of residue tasks in these subintervals. The eighth and ninth columns represent U_{RMN} and U_{RMX}, respectively. The rows of the table are divided into three categories, based on three value ranges of U_{RMN} as in the following equations:

$$\begin{array}{ll} \textbf{CAT-0 (row no 1–20):} & 0 < U_{\mathrm{RMN}} \leq \frac{4Q}{3} \\ \textbf{CAT-1 (row no 21–41):} & \frac{4Q}{3} < U_{\mathrm{RMN}} \leq \frac{8Q}{3} \\ \textbf{CAT-2 (row no 42–44):} & \frac{8Q}{3} < U_{\mathrm{RMN}} \leq 4Q \end{array} \tag{5}$$

We now present the task-assignment algorithms for the three categories of residue tasks in I_2–I_6 and the residue tasks in I_7, whose collective purpose is to guarantee that, *if* $U_{RT} \leq \frac{4Qm''}{3}$ *for the smallest non-negative integer* m''*, all residue tasks are assigned to at most* m'' *processors.*

7.1 Residue Task Assignment: CAT-0

Consider the first 20 possibilities (rows 1–20 in Table 2) of CAT-0 residue tasks and all residue task in I_7. Assign tasks to processor as follows. *If* $U_{RT} \leq \frac{4Q}{3}$*, all residue tasks in* I_2–I_7 *are assigned to one processor. If* $U_{RT} > \frac{4Q}{3}$*, all residue tasks in* I_7 *are assigned to one processor and all CAT-0 residue tasks in* I_2–I_6 *are assigned to another processor.*

We prove the RM schedulability by considering two cases—case (i): $U_{\mathrm{RT}} \leq \frac{4Q}{3}$, and case (ii): $U_{\mathrm{RT}} > \frac{4Q}{3}$. If case (i) is true, all residue tasks in I_2–I_7 are assigned to one processor. Since $U_{\mathrm{RT}} \leq \frac{4Q}{3} < LLB(\infty) = \ln 2$, all residue tasks are RM schedulable on one processor. If case (ii) applies, all residue tasks in I_7 assigned to one processor are RM schedulable using Lemma 1. Next, the CAT-0 residue tasks in I_2–I_6 are assigned to another processor. According to column seven of Table 2, the number of CAT-0 residue tasks in I_2–I_6 is at most 3. And observing the

No.	I_2	I_3	I_4	I_5	I_6	Total	U_{RMN}	U_{RMX}
CAT-0								
1	0	0	0	0	1	1	$\frac{Q}{3}$	$\frac{4Q}{9}$
2	0	0	0	1	0	1	$\frac{4Q}{9}$	$\frac{8Q}{15}$
3	0	0	1	0	0	1	$\frac{8Q}{15}$	$\frac{2Q}{3}$
4	0	1	0	0	0	1	$\frac{2Q}{3}$	$\frac{8Q}{9}$
5	0	0	0	0	2	2	$\frac{2Q}{3}$	$\frac{8Q}{9}$
6	0	0	0	1	1	2	$\frac{7Q}{9}$	$\frac{44Q}{45}$
7	0	0	0	2	0	2	$\frac{8Q}{9}$	$\frac{16Q}{15}$
8	0	0	1	0	1	2	$\frac{13Q}{15}$	$\frac{10Q}{9}$
9	0	0	1	1	0	2	$\frac{44Q}{45}$	$\frac{18Q}{15}$
10	0	1	0	0	1	2	Q	$\frac{4Q}{3}$
11	1	0	0	0	0	1	$\frac{8Q}{9}$	$\frac{4Q}{3}$
12	0	0	0	0	3	3	Q	$\frac{4Q}{3}$
13	0	0	2	0	0	2	$\frac{16Q}{15}$	$\frac{4Q}{3}$
14	0	1	0	1	0	2	$\frac{10Q}{9}$	$\frac{64Q}{45}$
15	0	0	0	1	2	3	$\frac{10Q}{9}$	$\frac{64Q}{45}$
16	0	0	0	2	1	3	$\frac{11Q}{9}$	$\frac{68Q}{45}$
17	0	1	1	0	0	2	$\frac{18Q}{15}$	$\frac{14Q}{9}$
18	0	0	1	0	2	3	$\frac{18Q}{15}$	$\frac{14Q}{9}$
19	0	0	1	1	1	3	$\frac{59Q}{45}$	$\frac{74Q}{45}$
20	1	0	0	0	1	2	$\frac{11Q}{9}$	$\frac{16Q}{9}$
CAT-1								
21	0	0	0	1	3	4	$\frac{13Q}{9}$	$\frac{28Q}{15}$
22	0	1	0	2	0	3	$\frac{14Q}{9}$	$\frac{88Q}{45}$
23	0	0	0	2	2	4	$\frac{14Q}{9}$	$\frac{88Q}{45}$
24	0	0	1	0	3	4	$\frac{23Q}{15}$	2Q
25	0	0	3	0	0	3	$\frac{24Q}{15}$	2Q
26	0	1	1	0	1	3	$\frac{23Q}{15}$	2Q
27	0	1	1	1	0	3	$\frac{74Q}{45}$	$\frac{94Q}{45}$
28	0	0	1	1	2	4	$\frac{74Q}{45}$	$\frac{94Q}{45}$
29	1	1	0	0	0	2	$\frac{14Q}{9}$	$\frac{20Q}{9}$
30	0	1	2	0	0	3	$\frac{26Q}{15}$	$\frac{20Q}{9}$
31	1	0	0	0	2	3	$\frac{14Q}{9}$	$\frac{20Q}{9}$
32	0	0	0	2	3	5	$\frac{17Q}{9}$	$\frac{12Q}{5}$
33	0	0	1	1	3	5	$\frac{89Q}{45}$	$\frac{38Q}{15}$
34	1	0	0	0	3	4	$\frac{17Q}{9}$	$\frac{8Q}{3}$
35	0	0	4	0	0	4	$\frac{32Q}{15}$	$\frac{8Q}{3}$
36	1	1	0	0	1	3	$\frac{17Q}{9}$	$\frac{8Q}{3}$
37	2	0	0	0	0	2	$\frac{16Q}{9}$	$\frac{8Q}{3}$
38	2	0	0	0	1	3	$\frac{19Q}{9}$	$\frac{28Q}{9}$
39	2	1	0	0	0	3	$\frac{22Q}{9}$	$\frac{32Q}{9}$
40	0	1	3	0	0	4	$\frac{34Q}{15}$	$\frac{26Q}{9}$
41	2	0	0	0	2	4	$\frac{22Q}{9}$	$\frac{32Q}{9}$
CAT-2								
42	2	1	0	0	1	4	$\frac{25Q}{9}$	4Q
43	2	0	0	0	3	5	$\frac{25Q}{9}$	4Q
44	0	1	4	0	0	5	$\frac{14Q}{5}$	$\frac{32Q}{9}$

Table 2: All 44 possibilities of Residue tasks in I_2–I_6

ninth column, we find that the maximum total utilization (U_{RMX}) of all CAT-0 residue tasks in I_2–I_6 for any row 1–20 is $\frac{16Q}{9} \approx 0.736$ (see row 20). Since $\frac{16Q}{9} < LLB(3) \approx 0.779$, all CAT-0 residue tasks from I_2–I_6 are RM schedulable on the second processor. In summary, *for CAT-0 residue tasks, if* $U_{RT} \leq \frac{4Q}{3}$*, we need one processor, otherwise, we need at most two processors to assign all the residue tasks in* I_2–I_7.

7.2 Residue Task Assignment: CAT-1

Before we propose the task assignment algorithms for this category, consider the following Theorem from [13] that is used to assign total t tasks in s processors using RMFF algorithm[3].

Theorem 1 (from [13])**.** *All t tasks in set A are schedulable on s processors using RMFF, if* $U(A) \leq (s-1)Q + (t-s+1)(2^{\frac{1}{t-s+1}} - 1)$.

We denote the bound in Theorem 1 by $U(s,t) = (s-1)Q + (t-s+1)(2^{\frac{1}{t-s+1}} - 1)$. Now we consider the next 21 possibilities (rows 21–41 in Table 2) of CAT-1 residue tasks and residue tasks in I_7. If $U_{\text{RT}} \leq \frac{8Q}{3}$, we assign all CAT-1 residue tasks in I_2–I_6 for rows 21–41 and the residue tasks in I_7 to at most two processors, otherwise, to at most three processors.

For the first case, *if* $U_{RT} \leq \frac{8Q}{3}$*, all CAT-1 residue tasks in* I_2–I_7 *are assigned using RMFF allocation to two processors.* We prove the RM schedulability in Lemma 2.

Lemma 2. *If* $U_{RT} \leq \frac{8Q}{3}$*, then all the CAT-1 residue tasks in* I_2–I_6 *and the residue tasks in* I_7 *are RM schedulable on two processors using FF allocation.*

Proof. According to Theorem 1, $U(s,t)$ for s=2 is $U(2,t) = Q + (t-1)(2^{\frac{1}{(t-1)}} - 1)$. Note that, for rows 21–41, the number of residue tasks $t \geq 2$. The function $U(2,t)$ is monotonically non-increasing as t increases. The minimum of $U(2,t)$ is $Q + \ln 2$ as $t \to \infty$. Therefore, $U(2,t) \geq Q + \ln 2 = 1.10736$ for any t. Since $U_{\text{RT}} \leq \frac{8Q}{3} = 1.10456$, we have $U_{\text{RT}} < U(2,t)$. Using Theorem 1, all CAT-1 residue tasks in I_2–I_6 and the residue tasks in I_7 are RM schedulable on two processors if $U_{\text{RT}} \leq \frac{8Q}{3}$. □

For the second case, *if* $U_{RT} > \frac{8Q}{3}$*, we assign all residue tasks in* I_7 *to one processor (RM schedulable using Lemma 1). And, all residue tasks in* I_2–I_6 *are assigned to at most two processors using algorithms* `R21_37` *,* `R38` *,* `R39` *, and* `R40_41` *for row 21–37, row 38, row 39 and row 40–41 in Table 2, respectively.* Next, we present each of these algorithms and show that all CAT-1 residue tasks in I_2–I_6 are RM schedulable on at most two processors.

Algorithm `R21_37` **:** All residue tasks in I_2–I_6 for rows 21–37 are assigned to two processors using FF allocation. Such tasks are RM schedulable using Lemma 3.

Lemma 3. *All the residue tasks in* I_2–I_6 *given in any row of 21–37 of Table 2 are RM schedulable on two processors using FF allocation.*

Proof. According to column seven in Table 2 for rows 21–37, the number of residue tasks in I_2–I_6 is at most 5. Therefore, $U(2,t)$ is minimized for rows 21–37 when $t = 5$, and we have, $U(2,5) = Q + 4(2^{\frac{1}{4}} - 1) \approx 1.17$. Observing the ninth column of rows 21–37, we find that the maximum total utilization (U_{RMX}) of the residue tasks in I_2–I_6 is at most $\frac{8Q}{3} \approx 1.105$ (see row 37). Since $\frac{8Q}{3} < U(2,5)$, all the $t \leq 5$ residue tasks in I_2–I_6 for any row 21–37 are RM schedulable on two processors using FF allocation. □

Algorithm `R38` **:** For row 38, there are two tasks in I_2 and one task in I_6. Assume that $\tau_a \in I_2$, $\tau_b \in I_2$ and $\tau_c \in I_6$. Task $\tau_a \in I_2$ is assigned to one dedicated processor and therefore trivially RM schedulable. Then, $\tau_b \in I_2 = (\frac{8Q}{9}, \frac{4Q}{3}]$ and $\tau_c \in I_6 = (\frac{Q}{3}, \frac{4Q}{9}]$ are assigned to another

[3]Symbols n and m in [13] are renamed as s and t in this paper for clarity.

processor. Since $u_b \leq \frac{4Q}{3}$ and $u_c \leq \frac{4Q}{9}$, we have $u_b + u_c \leq \frac{4Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < 2Q = LLB(2)$. So, τ_b and τ_c are schedulable on one processor. All three residue tasks in row 38 are thus RM schedulable on two processors.

Algorithm `R39` **:** For row 39, there are two tasks in I_2 and one task in I_3. Assume that $\tau_a \in I_2$, $\tau_b \in I_2$ and $\tau_c \in I_3$. We assign these tasks by calling SPLIT ($\{\tau_a,\tau_b,\tau_c\}$) where the highest-priority task is split. We prove that task τ_a, τ_b and τ_c are RM schedulable on two processors in Lemma 4.

Lemma 4. *If $\tau_a \in I_2$, $\tau_b \in I_2$ and $\tau_c \in I_3$, then all three tasks are RM schedulable in two processors using* SPLIT *($\{\tau_a,\tau_b,\tau_c\}$).*

Proof. It has already been proven (Policy 2 in Section 5) that three tasks in I_2 are RM schedulable on two processors using SPLIT . The utilization of a task from I_3 is smaller than that of a task in I_2. Hence, two task from I_2 and one from I_3 are also RM schedulable using SPLIT on two processors. □

Algorithm `R40_41` **:** Four residue tasks either in row 40 or in row 41 are scheduled on two processors as in Fig. 3. We prove the RM schedulability of these tasks in Lemma 5.

Algorithm `R40_41` (Residue tasks for row 40 or 41)

1. Select τ_a and τ_b from two different subintervals
2. Let τ_c and τ_d be the remaining residue tasks
3. Assign τ_a, τ_b to one processor
4. Assign τ_c, τ_d to one processor

Figure 3: Residue Task Assignment (row 40 or row 41)

Lemma 5. *The residue tasks in row 40 or 41 in Table 2 are RM schedulable on two processors using algorithm* `R40_41` *.*

Proof. The four residue tasks either in row 40 or row 41 are from exactly two subintervals. For row 40, there is one residue task in $I_3=(\frac{2Q}{3}, \frac{8Q}{9}]$ and three residue tasks in $I_4=(\frac{8Q}{15}, \frac{2Q}{3}]$. For row 41, there are two residue tasks in each of $I_2=(\frac{8Q}{9}, \frac{4Q}{3}]$ and $I_6=(\frac{Q}{3}, \frac{4Q}{9}]$. Now, consider two cases: case (i) for row 40 and case (ii) for row 41.

Case (i): Two tasks from two different subintervals of row 40 (line 1) satisfy $\tau_a \in I_3$ and $\tau_b \in I_4$. We then have $\tau_c \in I_4$ and $\tau_d \in I_4$ (line 2). So, $u_a \leq \frac{8Q}{9}$, $u_b \leq \frac{2Q}{3}$, $u_c \leq \frac{2Q}{3}$ and $u_d \leq \frac{2Q}{3}$. Task τ_a and τ_b are assigned to one processor (line 3); we have $u_a + u_b \leq \frac{8Q}{9} + \frac{2Q}{3} = \frac{14Q}{9} < 2Q = LLB(2)$. Task τ_c and τ_d are assigned to one processor (line 4); we have $u_c + u_d \leq \frac{2Q}{3} + \frac{2Q}{3} = \frac{4Q}{3} < 2Q = LLB(2)$. Thus, all residue tasks in row 40 are RM schedulable on two processors.

Case (ii): Two tasks from two different subintervals of row 41 (line 1), satisfy $\tau_a \in I_2$ and $\tau_b \in I_6$. We then have $\tau_c \in I_2$ and $\tau_d \in I_6$ (line 2). So, $u_a \leq \frac{4Q}{3}$, $u_b \leq \frac{4Q}{9}$, $u_c \leq \frac{4Q}{3}$ and $u_d \leq \frac{4Q}{9}$. Task τ_a and τ_b are assigned to one processor (line 3); we have $u_a + u_b \leq \frac{4Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < 2Q = LLB(2)$. Similarly, task τ_c and τ_d are assigned to one processor (line 4); we have $u_c + u_d \leq LLB(2)$. Thus, all residue tasks in row 41 are RM schedulable on two processors. □

In summary, for CAT-1 residue tasks *If $U_{RT} \leq \frac{8Q}{3}$, then* `IBPS` *needs to assign all residue tasks in I_2–I_7 to at most two processors, and otherwise* `IBPS` *needs at most three processors.*

7.3 Residue Task Assignment: CAT-2

We now consider the last three possibilities (rows 42–44 in Table 2) of CAT-2 residue tasks in I_2–I_6 and the residue tasks in I_7. We propose two task assignment algorithms `R42` and `R43_44` for residue tasks in row 42 and rows 43–44, respectively, along with all residue tasks in I_7.

Algorithm `R42` **:** Algorithm `R42` (see Fig. 4) assigns the four CAT-2 residue tasks in row 42 and residue tasks in I_7. The RM schedulability using algorithm `R42` is ensured as follows.

Algorithm `R42` (Residue task in I_2–I_7 in row 42)

1. Let $\tau_a \in I_2$, $\tau_b \in I_2$, $\tau_c \in I_3$ and $\tau_d \in I_6$
2. SPLIT ($\{\tau_a,\tau_b,\tau_c\}$)
3. **if** ($U_{\text{RT}} \leq 4Q$) **then**
4. Assign τ_d and all tasks of I_7 to one processor
5. **else**
6. Assign τ_d to one processor.
7. Assign all tasks of I_7 (if any) to one processor
8. **end if**

Figure 4: Residue Assignment(row 42)

There are four residue tasks in row 42 such that $\tau_a \in I_2$, $\tau_b \in I_2$, $\tau_c \in I_3$ and $\tau_d \in I_6$ (line 1 in Fig. 4). Tasks τ_a, τ_b and τ_c are assigned to two processors (line 2) by calling SPLIT ($\{\tau_a,\tau_b,\tau_c\}$). Such three tasks are RM schedulable on two processors according to Lemma 4. Next, residue task $\tau_d \in I_6$ and all residue tasks in I_7 are assigned to one or two processors depending on two cases: case (i) $U_{\text{RT}} \leq 4Q$ and, case (ii) $U_{\text{RT}} > 4Q$ respectively.

Case (i) $U_{\text{RT}} \leq 4Q$**:** When $U_{\text{RT}} \leq 4Q$ (line 3), task τ_d and all residue tasks in I_7 are assigned to a third processor (line 4). So, to ensure RM schedulability on one processor, we prove that, $u_d + U(S) \leq LLB(\infty) \approx 0.693$ where S is the set of residue tasks in I_7. For row 42, we have $U_{\text{RT}} = u_a+u_b+u_c+u_d+U(S)$ and $U_{\text{RMN}} < u_a+u_b+u_c+u_d$. Therefore, $U_{\text{RT}} > U_{\text{RMN}}+U(S)$. Since, for row 42, $U_{\text{RMN}} = \frac{25Q}{9}$ (see eighth column of row 42) and $U_{\text{RT}} \leq 4Q$ (case assumption), we have $U(S) < 4Q - \frac{25Q}{9} = \frac{11Q}{9}$. Since $\tau_d \in I_6 = (\frac{Q}{3}, \frac{4Q}{9}]$, we have $u_d+U(S) \leq \frac{4Q}{9}+\frac{11Q}{9} = \frac{15Q}{9} \approx 0.69035$. Therefore, $u_d + U(S) \leq LLB(\infty) \approx 0.693$.

Case (ii) $U_{\text{RT}} > 4Q$**:** When $U_{\text{RT}} > 4Q$, task τ_d is assigned to the third processor (line 6), which is trivially RM schedulable. All residue tasks in I_7 are scheduled on another processor (line 7), which are also RM schedulable according to Lemma 1. So, task τ_d and all residue tasks in I_7 are RM schedulable on two processor for row 42 whenever $U_{\text{RT}} > 4Q$.

In summary, *If* $U_{RT} \leq 4Q$*, the four CAT-2 residue tasks in row 42 and all residue tasks in* I_7 *are RM schedulable on at most three processors; otherwise, the tasks are schedulable on at most four processors using* `R42`*.*

Algorithm `R43_44` **:** Algorithm `R43_44` (see Fig. 5) assigns the five CAT-2 residue tasks in row 43 or row 44 and residue tasks in I_7.

The RM schedulability using algorithm `R43_44` is ensured as follows. The five tasks in row 43 or in row 44 are denoted by τ_a, τ_b, τ_c, τ_d and τ_e. Tasks τ_a and τ_b are from two different subintervals (line 1). For row 43, there are two residue tasks in $I_2=(\frac{8Q}{9}, \frac{4Q}{3}]$ and three residue tasks in $I_6=(\frac{Q}{3}, \frac{4Q}{9}]$. For row 44, there is one residue task in $I_3=(\frac{2Q}{3}, \frac{8Q}{9}]$ and four residue tasks in $I_4=(\frac{8Q}{15}, \frac{2Q}{3}]$. Tasks τ_a and τ_b are assigned to one processor (line 4). For row 43, $\tau_a \in I_2$ and $\tau_b \in I_6$ and we have $u_a + u_b \leq \frac{4Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < LLB(2)$. For row 44, $\tau_a \in I_3$ and $\tau_b \in I_4$, we have $u_a + u_b \leq \frac{8Q}{9} + \frac{2Q}{3} = \frac{14Q}{9} < LLB(2)$. So, tasks τ_a and τ_b are RM schedulable on one processor for any row 43 or 44. To prove the RM schedulability of τ_c, τ_d, τ_e and all residue tasks in I_7, we consider two cases: case (i): $u_c + u_d + u_e \leq LLB(3)$ (line 5), case (ii): $u_c + u_d + u_e > LLB(3)$ (line 8).

Case (i): Here, τ_c, τ_d and τ_e are assigned to a second processor (line 6) and they are RM

Algorithm R43_44 (Residue tasks for row 43 or 44)

1. Select τ_a and τ_b in two different subintervals of I_2–I_6
2. Let τ_c, τ_d and τ_e be the remaining tasks in I_2–I_6
3. such that, $u_c \geq u_d$ and $u_c \geq u_e$
4. Assign τ_a and τ_b to one processor.
5. **if** $(U(\tau_c) + U(\tau_d) + U(\tau_e)) \leq LLB(3)$ **then**
6. Assign τ_c, τ_d and τ_e to a processor.
7. Assign all tasks of I_7 (if any) to a processor
8. **else**
9. Assign τ_c and τ_d to a processor.
10. **if** $(U_{\mathrm{RT}} \leq 4Q)$ **then**
11. Assign τ_e and all tasks of I_7 to a processor
12. **else**
13. Assign τ_e to one processor.
14. Assign all tasks of I_7 (if any) to a processor
15. **end if**
16. **end if**

Figure 5: Residue Assignment (row 43-44)

schedulable (case assumption). All residue tasks in I_7 are assigned to a third processor (line 7) and are RM schedulable using Lemma 1.

Case (ii): Here, τ_c and τ_d are assigned to a second processor (line 9). For row 43, $\tau_c \in I_2 = (\frac{8Q}{9}, \frac{4Q}{3}]$ since $u_c \geq u_d$ and $u_c \geq u_e$ (line 3). Then obviously, $\tau_d \in I_6 = (\frac{Q}{3}, \frac{4Q}{9}]$ for row 43. We have $u_c + u_d \leq \frac{4Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < LLB(2)$. For row 44, both τ_c and τ_d are in I_4=$(\frac{8Q}{15}, \frac{2Q}{3}]$, and we have $u_c + u_d \leq \frac{2Q}{3} + \frac{2Q}{3} = \frac{4Q}{3} < LLB(2)$. So, τ_c and τ_d assigned to one processor (line 9) are RM schedulable for row 43 or 44. Next, task τ_e and all residue tasks in I_7 are scheduled on one or two processors depending on two subcases: subcase (i): $U_{RT} \leq 4Q$ (line 10) and, subcase (ii): $U_{RT} > 4Q$ (line 12).

Subcase(i): When $U_{RT} \leq 4Q$, task τ_e and all residue tasks in I_7 are assigned to a third processor (line 11). For RM schedulability, we show that $u_e + U(S) \leq LLB(\infty) = \ln 2$ where S is the set of residue tasks in I_7. Note that $U_{\mathrm{RMN}} + U(S) < U_{\mathrm{RT}} \leq 4Q$ for this subcase. Since, for row 43, $U_{\mathrm{RMN}} = \frac{25Q}{9}$ (see column 8), we have $U(S) \leq 4Q - \frac{25Q}{9} = \frac{11Q}{9}$. Since $\tau_e \in I_6 = (\frac{Q}{3}, \frac{4Q}{9}]$ for row 43, we have $u_e + U(S) \leq \frac{4Q}{9} + \frac{11Q}{9} = \frac{15Q}{9} \approx 0.6903 < \ln 2$.

For row 44, since $\tau_a \in I_3 = (\frac{2Q}{3}, \frac{8Q}{9}]$, $\tau_b \in I_4 = (\frac{8Q}{15}, \frac{2Q}{3}]$ and $u_c + u_d + u_e > LLB(3) = 3(2^{\frac{1}{3}} - 1)$ for case (ii) we have, $u_a + u_b + u_c + u_d + u_e > \frac{2Q}{3} + \frac{8Q}{15} + 3(2^{\frac{1}{3}} - 1) = \frac{6Q}{5} + 3(2^{\frac{1}{3}} - 1)$. Since $U(S) + u_a + u_b + u_c + u_d + u_e = U_{\mathrm{RT}}$ and $U_{\mathrm{RT}} \leq 4Q$ (subcase assumption), we have $U(S) \leq 4Q - (\frac{6Q}{5} + 3(2^{\frac{1}{3}} - 1)) = \frac{14Q}{5} - 3(2^{\frac{1}{3}} - 1)$. Since $\tau_e \in I_4 = (\frac{8Q}{15}, \frac{2Q}{3}]$ for row 44, we have $u_e + U(S) \leq \frac{2Q}{3} + \frac{14Q}{5} - 3(2^{\frac{1}{3}} - 1) \approx 0.6561 < \ln 2 = LLB(\infty)$. So, τ_e and all residue tasks from I_7 are schedulable on one processor for row 43 or row 44 if $U_{RT} \leq 4Q$. So, in order to assign τ_a, τ_b, τ_c, τ_d and τ_e and residue tasks in I_7 we need at most three processors if $U_{RT} \leq 4Q$.

Subcase(ii): When $U_{RT} > 4Q$, task τ_e is assigned to a third processor (line 13) and is trivially RM schedulable. All residue tasks in I_7 are assigned to a fourth processor (line 14) and RM schedulable using Lemma 1. So, in order to assign τ_a, τ_b, τ_c, τ_d and τ_e and residue tasks in I_7, we need at most four processors if $U_{RT} > 4Q$.

In summary, *if $U_{RT} \leq 4Q$, we assign residue tasks in I_2–I_6 and residue tasks for I_7 to at most three processors; otherwise, these residue tasks are assigned to at most four processors.*

From the scheduling analysis in this section, we have the following fact.

Fact-1. Any CAT-x residue tasks, for $x = 0, 1, 2$, and residue tasks from I_7 are RM schedulable on at most $(x+1)$ processors if $U_{\mathrm{RT}} \leq \frac{4Q(x+1)}{3}$; otherwise, these residue tasks are schedulable on at most $(x+2)$ processors. We have the following Theorem 2.

Theorem 2. *If $U_{RT} \leq \frac{4Qm''}{3}$, for the smallest non-negative integer m'', all the residue tasks are RM schedulable on at most m'' processors.*

Proof. Note that, if residue tasks only exist in I_7, our theorem is true because of Lemma 1. Now, consider any CAT-x residue tasks and residue tasks from I_7. For CAT-x residue tasks, we have $U_{\text{RT}} > U_{\text{RMN}} > \frac{4Qx}{3}$ using Eq. (4)–(5). If $U_{\text{RT}} \leq \frac{4Q(x+1)}{3}$, then all the residue tasks are RM schedulable on $(x+1)$ processors (using Fact-1). Note that $m'' = (x+1)$ is the smallest non-negative integer such that $U_{RT} \leq \frac{4Qm''}{3}$. Now, if $U_{\text{RT}} > \frac{4Q(x+1)}{3}$, then $U_{\text{RT}} \leq \frac{4Qm''}{3}$ for some $m'' \geq (x+2)$. Since, using Fact-1 in such case, all residue tasks are assigned to at most $(x+2)$ processors, our theorem is true for the smallest nonnegative integer $m'' \geq (x+2)$ such that $U_{\text{RT}} \leq \frac{4Qm''}{3}$. □

Task assignment to processor completes here. The task assignment algorithms in this phase also runs in linear time. The run time dispatcher of `IBPS` is given in Appendix C.

8 Performance of `IBPS`:

Utilization Bound: The worst-case utilization bound of `IBPS` is given in Theorem 3.

Theorem 3. *If $U(\Gamma) \leq \frac{4Qm}{3}$, all tasks meet deadlines on at most m processors using `IBPS`.*

Proof. As shown in Section 5–6, each processor, to which tasks have been assigned during the first two phases, will have an RM schedulable task set with a total utilization strictly greater than $\frac{4Q}{3}$ due to load regulation. Let $m' \geq 0$ be the number of processors to which tasks are assigned during the first two phases. Theorem 2 states that, if $U_{RT} \leq \frac{4Qm''}{3}$ for the smallest non-negative integer m'', then all residue tasks are RM schedulable on at most m'' processors during the third phase. We must show that, if $U(\Gamma) \leq \frac{4Qm}{3}$, then $(m' + m'') \leq m$. Since $U_{RT} \leq \frac{4Qm''}{3}$ for the smallest integer m'', we have $\frac{4Q(m''-1)}{3} < U_{\text{RT}}$. The total utilization of tasks assigned to m' processors during the first two phases is $U(\Gamma) - U_{\text{RT}}$. Therefore, $U(\Gamma) - U_{\text{RT}} > \frac{4Qm'}{3}$ and we have, $\frac{4Q(m''-1)}{3} + \frac{4Qm'}{3} < U(\Gamma)$. If $U(\Gamma) \leq \frac{4Qm}{3}$, then $\frac{4Q(m'+m''-1)}{3} < \frac{4Qm}{3}$ which implies $(m' + m'' - 1) < m$. Because m', m'', and m are non-negative integers, we have $(m' + m'') \leq m$. So, if $U(\Gamma) \leq \frac{4Qm}{3}$, all tasks in set Γ are RM schedulable on at most m processors. Since $Q = (\sqrt{2} - 1)$, the utilization bound on m processors is $\frac{4(\sqrt{2}-1)m}{3} \approx 55.2\%$. □

Resource Augmentation: Resource augmentation compares a given algorithm against an optimal algorithm by determining the factor by which if a given multiprocessor platform is augmented, then the given algorithm has equal performance to the optimal. In this paper, we find the resource augmentation factor for `IBPS` as follows: given a task set Γ known to be feasible on m processors each having speed ζ, we determine the multiplicative factor of this speed by which the platform of `IBPS` can be augmented so that Γ is schedulable using `IBPS`. Baruah and Fisher in [23] have proved that, *if task system Γ is feasible (under either partitioned or global paradigm) on an identical multiprocessor platform comprised of m processors each with speed ζ, then we must have $m\zeta \geq U(\Gamma)$.* Now, if $\frac{4Q}{3} \geq \zeta$, then $m \geq \frac{3m\zeta}{4Q}$. Using the necessary condition $m\zeta \geq U(\Gamma)$ for feasibility in [23], we have $m \geq \frac{3U(\Gamma)}{4Q} \Leftrightarrow U(\Gamma) \leq \frac{4Qm}{3}$. According to Theorem 3, Γ is schedulable on m unit-capacity processors. Therefore, the processor speed-up factor for `IBPS` is $\frac{1}{\zeta} \geq \frac{3}{4Q} \approx 1.81$.

9 Admission Controller `O-IBPS`

In this section we present an efficient admission controller for online task scheduling, called `O-IBPS`. When a multiprocessor scheduling algorithm is used on-line, the challenge is to determine

how a new on-line task τ_{new} is accepted and assigned to a processor. In O-IBPS, *if* $U(\Gamma) + u_{new} \leq \frac{4Qm}{3}$*, the new task* τ_{new} *is accepted to the system.* Theorem 3 ensures that, we have sufficient capacity to assign the new task τ_{new} using IBPS. If $u_{new} \in \mathrm{I}_1$, we assign this new task to a dedicated processor. Otherwise, if $\tau_{new} \in \mathrm{I}_k$ for some $k = 2, 3, \ldots 7$, we have $u_{new} \leq \frac{4Q}{3}$. Let Γ_{R} denote the set of residue tasks before τ_{new} is assigned to a processor such that $\frac{4Q(m''-1)}{3} \leq U(\Gamma_R) \leq \frac{4Qm''}{3}$. These residue tasks in Γ_R were assigned on at most m'' processors (using Theorem 2) before τ_{new} arrives to the system. O-IBPS then forms a new task set $\Gamma_{\mathrm{new}} = \Gamma_R \cup \{\tau_{new}\}$. If $U(\Gamma_{new}) \leq \frac{4Qm''}{3}$, Γ_{new} is assigned to m'' processors using the IBPS task assignment phases. If $U(\Gamma_{new}) > \frac{4Qm''}{3}$, then we have, $U(\Gamma_{new}) = U_{\mathrm{R}} + u_{new} \leq \frac{4Qm''}{3} + \frac{4Q}{3} = \frac{4Q(m''+1)}{3}$. Γ_{new} is assigned to at most $(m'' + 1)$ processors using IBPS (one new processor is introduced).

When a task leaves the system, say from processor Θ_x, then for load regulation we re-execute the assignment algorithm on Θ_x plus on all m'' processors. Since residue tasks never require more than four processors (See Section 7)during third phase of IBPS, we have $m'' \leq 4$. Thus, when τ_{new} is admitted to the system using O-IBPS, the number of processors that require reassignment of task is upper bounded by $min\{4, m\}$. And when a task leaves the system, the number of processors that require reassignment of task for load regulation is upper bounded by $min\{5, m\}$. Remember that, IBPS runs in linear time. And the trend in processor industry is to have chip multiprocessor with many cores (16, 32, 64 cores or even higher) in near future. So, our scheduling algorithm is efficient and scalable with increasing number of cores in CMPs for online scheduling of real-time tasks.

10 Related Work

Non task splitting algorithms can not have utilization bound greater than 50%. In [15], it is shown that the worst case utilization bound for partitioned RM First-Fit (FF) scheduling is $m(\sqrt{2} - 1) \approx 41\%$. In [13], this bound is improved by also including the number of tasks in the schedulability condition. In [10], an algorithm R-BOUND-MP-NFR (based on the R-BOUND test in [14]) is proposed that has a utilization bound of 50%. The work in [11, 12] assigns tasks to processors according to FF with a decreasing deadline order and their worst-case performance is characterized using resource augmentation.

In order to achieve a utilization bound for the partitioned approach that exceeds 50%, a new type of scheduling algorithms using task splitting has evolved [16, 17, 21, 18, 19, 24, 20]. Most of these works address task splitting for dynamic priority. In [16], a task splitting algorithm EDF-fm is proposed that has no scheduling guarantee but instead offers bounded task tardiness. An algorithm, called EKG [17], for dynamic-priorities using task splitting has a utilization bound between 66% and 100% depending on a design parameter k which trade-off utilization bound and preemption count. Using a time slot-based technique, sporadic task scheduling for constrained and arbitrary deadline are developed in [21, 18]. Using the dynamic-priority algorithm EDDP in [19], the deadline of a split task is changed to a smaller deadline called 'virtual deadline'. Common for all these dynamic-priority task splitting algorithms is the absence of priority traceability property and load regulation. As many of the algorithms requires sorting task before assignment, online scheduling may be inefficient. In [20], an implicit deadline task set is converted to a constrained deadline static-priority task set during task assignment. Even if this algorithm has more than 50% utilization bound, it does not have priority traceability property and does not consider its online applicability as in IBPS. The static-priority task splitting algorithm in [24, 25] has utilization bound that does not exceed 50%.

11 Conclusion

In this paper, we propose a task assignment algorithm, called IBPS, based on utilization of static-priority tasks in different subintervals having worst-case utilization bound 55.2%. The load regulation technique of IBPS enable designing of efficient admission controller for on-line task scheduling in multiprocessor system. With increasing number of processors in a multiprocessor system, the percentage of processors having load greater than 55.2% also increases since at most four processor could have load less than 55.2%. Therefore, online scheduling of tasks using O-IBPS scales with the current trend to have increasing number of cores in chip multiprocessors. Our algorithm possesses priority traceability property which facilitates the system designer's ability to debug and maintain a system during development. The task splitting algorithm has lower number of migrations compared to any other task splitting algorithm for static and dynamic priority. All these salient features make our scheduling algorithm efficient for practical implementation for chip multiprocessors with increasing number of cores.

References

[1] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, March-April 2005.

[2] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance sparc cmt processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, 2009.

[3] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[4] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[5] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," *in Proc. of RTSS*, pp. 193–202, 2001.

[6] T. P. Baker, "An analysis of fixed-priority schedulability on a multiprocessor," *Real-Time Systems*, vol. 32, no. 1-2, pp. 49–71, 2006.

[7] S. Baruah and J. Goossens, "Rate-monotonic scheduling on uniform multiprocessors," *IEEE Trans. on Comput.*, vol. 52, no. 7, pp. 966–970, 2003.

[8] M. Bertogna, M. Cirinei, and G. Lipari, "New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors," *in Proc. of Conf. on Princ. of Dist. Syst.*, pp. 306–321, 2005.

[9] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Trans. on Comput.*, vol. 44, no. 12, pp. 1429–1442, 1995.

[10] B. Andersson and J. Jonsson, "The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%," *in Proc. of ECRTS*, pp. 33–40, 2003.

[11] N. Fisher, S. Baruah, and T. P. Baker, "The partitioned scheduling of sporadic tasks according to static-priorities," *in Proc. of ECRTS*, pp. 118–117, 2006.

[12] N. Fisher and S. Baruah, "The partitioned, static-priority scheduling of sporadic real-time tasks with constrained deadlines on multiprocessor platforms," *in Proc. of Conf. on Princ. of Dist. Sys.*, pp. 291–305, 2005.

[13] J. M. López, M. García, J. L. Díaz, and D. F. García, "Utilization bounds for multiprocessor rate-monotonic scheduling," *Real-Time Systems*, vol. 24, no. 1, pp. 5–28, 2003.

[14] S. Lauzac, R. Melhem, and D. Mossé, "An efficient rms admission control and its application to multiprocessor scheduling," *in Proc. of International Parallel Processing Symposium*, pp. 511–518, 1998.

[15] D.-I. Oh and T. P. Baker, "Utilization bounds for n-processor rate monotone scheduling with static processor assignment," *Real-Time Systems*, vol. 15, no. 2, pp. 183–192, 1998.

[16] J. H. Anderson, V. Bud, and U. C. Devi, "An edf-based scheduling algorithm for multiprocessor soft real-time systems," *in Proc. of ECRTS*, pp. 199–208, 2005.

[17] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," *in Proc. of RTCSA*, pp. 322–334, 2006.

[18] B. Andersson, K. Bletsas, and S. Baruah, "Scheduling arbitrary-deadline sporadic task systems on multiprocessors," *in Proc. of RTSS*, pp. 385–394, 2008.

[19] S. Kato and N. Yamasaki, "Portioned edf-based scheduling on multiprocessors," *in Proc. of International Conference on Embedded Software*, pp. 139–148, 2008.

[20] K. Lakshmanan, R. Rajkumar, and J. P. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," *in Proc. of ECRTS*, pp. 239 – 248, 2009.

[21] B. Andersson and K. Bletsas, "Sporadic multiprocessor scheduling with few preemptions," *in Proc. of ECRTS*, pp. 243–252, 2008.

[22] S. Kato and N. Yamasaki, "Real-time scheduling with task splitting on multiprocessors," *in Proc. of RTCSA*, pp. 441–450, 2007.

[23] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *in Proc. of RTSS*, 2005.

[24] S. Kato and N. Yamasaki, "Semi-partitining fixed-priority scheduling on multiprocessor," *in Proc. of RTAS*, pp. 23–32, 2009.

[25] ——, "Portioned static-priority scheduling on multiprocessors," *in Proc. of IPDPS*, pp. 1–12, 2008.

APPENDIX

A Why do we have seven utilization subintervals?

The seven utilization intervals result from our four different strategies for task assignment: (i) low number of subtasks per split task, (ii) low number of split tasks, (iii) assigning low number of tasks per processor[4], and (iv) load regulation.

Following these four task assignment strategies, we start by assigning only one task to one processor exclusively. To avoid assigning a task with very small utilization to one processor exclusively, we need to select a task that belongs to certain utilization subintervals. If `IBPS` has worst-case utilization bound U_w and load regulation try to maintain load on most processors beyond U_w, then one task with utilization greater than U_w is assigned to one processor exclusively. Thus, we obtain our first utilization interval which is (U_w, 1]. The exact value of U_w is determined when we assign more than one task having utilization less than U_w to one processor. When we try to assign two tasks with utilization less than U_w to one processor, we find that if these two tasks have equal utilization, then each task's utilization can not be greater than Q=($\sqrt{2}$–1) according to LLB(n=2). This implies $U_w \leq 41\%$ without task splitting technique. To achieve U_w greater than 50%, we split a task with utilization less than U_w in two subtasks each having same utilization. We gain no advantage by having unequal utilization for the two subtasks as individual task utilization is bounded from below and our strategy is to assign minimum number of tasks per processor. So, each subtask has utilization at most $\frac{U_w}{2}$. We assign one such subtask and a non-split task to one processor. For RM schedulability, we must have $U_w + \frac{U_w}{2} \leq 2Q$. This implies the value of $U_w \leq \frac{4Q}{3}$ and we get our first utilization subinterval I_1=($\frac{4Q}{3}$,1]. Thus, this interval also defines the maximum possible utilization bound of `IBPS`.

The overall goal with the task assignment is to achieve the maximum possible U_w, which is $\frac{4Q}{3}$ ≈55.2%. Moreover, for load regulation, we try to maintain load in most processors greater than $\frac{4Q}{3}$. When assigning two tasks (one subtasks and one split tasks) to one processor, the individual task utilization has to be bounded from below to maintain individual processor load greater than $\frac{4Q}{3}$. Simple arithmetic shows that, such individual task utilization has to be bounded from below by $\frac{8Q}{9}$. In such case, the subtask and the non-split task has utilization at least $\frac{4Q}{9}$ and $\frac{8Q}{9}$, respectively. Such a subtask and the non-split task assigned to one processor has load greater than $\frac{4Q}{9} + \frac{8Q}{9} = \frac{4Q}{3}$ and we obtain our second utilization subinterval which is ($\frac{8Q}{9}$,$\frac{4Q}{3}$].

Next we consider assigning task with utilization less than or equal to $\frac{8Q}{9}$. Two tasks with utilization less than or equal to $\frac{8Q}{9}$ are RM schedulable in one processor without splitting since $2 \times \frac{8Q}{9} < LLB(2)$. However, for load regulation we need to bound the load of each task utilization from below by $\frac{2Q}{3}$. In such case, the total utilization (load on a processor) of the two tasks is at least $2 \times \frac{2Q}{3} = \frac{4Q}{3}$ and we get our third utilization interval ($\frac{2Q}{3}$,$\frac{8Q}{9}$].

Next we consider assigning task with utilization less than $\frac{2Q}{3}$. At this stage, there is no way we can assign two tasks (with or without splitting) to one processor to have the processor load greater than $\frac{4Q}{3}$. Therefore, we decide to assign three tasks to one processor. Since we want to minimize number of split task, we first try to assign three tasks with utilization less than $\frac{2Q}{3}$ to one processor without splitting. Since $3 \times \frac{2Q}{3} > LLB(3)$, we can not assure RM schedulability of three tasks if they all have utilization equal to $\frac{2Q}{3}$. Therefore, we split the highest priority task (of the three selected tasks) in two subtasks. Note that, such a subtask's utilization is upper bounded by $\frac{Q}{3}$. We assign one such subtask and two non-split tasks to one processor. The subtask and the two non-split task have maximum total utilization $\frac{Q}{3} + \frac{2Q}{3} + \frac{2Q}{3} = \frac{5Q}{3} < LLB(3)$ and hence RM schedulable in one processor. However, for load regulation we need to bound the

[4]According to LLB, the RM scheduling on uniprocessor achieves higher utilization bound if number of tasks assigned to the processor is small [3].

load of each task utilization from below by $\frac{8Q}{15}$. In such case, the minimum utilization of the subtask and the non-split task are $\frac{4Q}{15}$ and $\frac{8Q}{15}$, respectively . Therefore, total utilization (load on a processor) of the three tasks (one subtask and two non-split tasks) is at least $\frac{4Q}{15}+\frac{8Q}{15}+\frac{8Q}{15}=\frac{4Q}{3}$ and we get our forth utilization interval $(\frac{8Q}{15},\frac{2Q}{3}]$.

Next we consider assigning task with utilization less than or equal to $\frac{8Q}{15}$. Three tasks with utilization less than or equal to $\frac{8Q}{15}$ are RM schedulable in one processor without splitting since $3\times\frac{8Q}{15} < LLB(3)$. However, for load regulation we need to bound the load of each task utilization from below by $\frac{4Q}{9}$. In such case, the total utilization (load on a processor) of the two tasks is at least $3\times\frac{4Q}{9}=\frac{4Q}{3}$ and we get our fifth utilization interval $(\frac{4Q}{9},\frac{8Q}{15}]$.

Next we consider assigning task with utilization less than or equal to $\frac{4Q}{9}$ without task splitting. As obvious from discussion in last paragraph, three tasks with utilization less than or equal to $\frac{4Q}{9}$ can not have total utilization greater than $\frac{4Q}{3}$ on one processor. So, for load regulation we consider assigning four tasks with utilization less than or equal to $\frac{4Q}{9}$ to one processor. Four tasks with utilization less than or equal to $\frac{4Q}{9}$ are RM schedulable in one processor without any splitting since $4\times\frac{4Q}{9} < LLB(4)$. However, for load regulation we need to bound the load of each task utilization from below by $\frac{Q}{3}$. In such case, the total utilization (load on a processor) of the four tasks is at least $4\times\frac{Q}{3}=\frac{4Q}{3}$ and we get our sixth utilization interval $(\frac{Q}{3},\frac{4Q}{9}]$.

Next, we consider assigning tasks with utilization less than $\frac{Q}{3}$. We observe an important inequality that is $(LLB(\infty)-\frac{Q}{3})=(\ln 2-\frac{Q}{3})>\frac{4Q}{3}$. This inequality suggest that, we can assign one by one task with utilization less than $\frac{Q}{3}$ to one processor until the load of that processor exceed $LLB(\infty)$. When no more tasks can be assigned to the processor using Liu and Layland sufficient test, we must have the load of the processor greater than $\frac{4Q}{3}$. So, we do not need to bound the utilization of the tasks from below for load regulation in this case. And we obtain our last utilization interval $(0,\frac{Q}{3}]$

In summary, at each stage of task assignment, we bound the utilization of tasks from below for load regulation by selecting a task from certain utilization interval, split a task in only two subtasks when task splitting is unavoidable and try to assign minimum number of tasks to one processor. The consequence is that, we derive the seven utilization intervals I_1-I_7 in Table 1.

B Number of possibilities of residue tasks in I_2-I_6

Remember that, there is no unassigned task in I_1 and the total utilization of the unassigned tasks in I_7 is not greater than $\frac{4Q}{3}$ after first phase (see Section 5). After second phase of task assignment using algorithm OddAssign, we may still have some unassigned odd tasks in I_2-I_6. These unassigned odd tasks after second phased are called residue tasks in I_2–I_6. After the first phase, I_2 has 0–2 odd tasks, I_3 has 0–1 odd task, I_4 has 0–4 odd tasks, I_5 has 0–2 odd tasks, and I_6 has 0–3 odd tasks (see Section 5). Odd tasks thus exist in subintervals I_2–I_6 as one of $(3 \times 2 \times 5 \times 3 \times 4 =)360$ possibilities after the first phase. During the second phase, algorithm OddAssign is able to assign *all* the odd tasks in subintervals I_2–I_6 for 316 out of the 360 possibilities after the first phase. For example, after first phase we might have 1 odd task in I_2 and 1 odd task in I_4 and no tasks in other subintervals for a task set. This is one of the 360 possibilities of odd tasks in I_2-I_6 after first phase. The first while loop in algorithm OddAssign[5] in Figure 6 would assign both of these unassigned odd tasks from I_2 and I_4 to one processor. So, after second phase no task remains unassigned for this possibility of odd task. Now consider another example where, after first phase, we might have 2 odd task in I_2 and 1 odd task in I_4 and no tasks in other subintervals of I_2-I_6 for a task set. This is another possibility of odd tasks out of the 360 possibilities of odd tasks after first phase. The first while loop in algorithm OddAssign in Figure 6 would assign one of the odd tasks in I_2 and one odd task in I_4 to one processor. But still one odd task in I_2 can not be assigned to any processor using OddAssign in second phase and has to be considered in third phase as residue task. This is one of the possibilities of residue tasks after second phase. For any task set, residue tasks thus exist, in subintervals I_2–I_6, as one of 44 possibilities after the second phase. In this appendix, we formally prove that, *residue tasks in subintervals* I_2–I_6 *that need to be handled in the third phase exist as one of the remaining 44 different possibilities for any task set.* In particular, the number of unassigned tasks in each subintervals I_2-I_6 after second phase for each of the 44 possibilities is determined.

To prove this fact, we would consider five different cases separately where odd tasks may exist in exactly any one, two, three, four or five of the subintervals I_2-I_6 after first phase. For each case, we determine the number of unassigned odd tasks in each subinterval for a possibility of odd task after second phase that become a possibility of residue tasks in third phase.

B.1 Case 1: Odd tasks exactly in one of the subintervals I_2-I_6

Remember that, the second phase does not assign tasks from only one subinterval, rather it assign tasks from more than one subintervals to processors. As a result, if after first phase, exactly one of the five subintervals I_2-I_6 has odd tasks and four other subintervals have no odd task, then second phase does not assign such odd task to processors. Such odd tasks are declared as residue tasks and are assigned to processors during the third phase. Now consider each subinterval separately. After first phase, number of tasks in subinterval I_2 can be either 1 or 2 (See Policy 1 in Section 5). So there are 2 possibilities when some odd tasks exits only in I_2. Similarly, number of odd task in I_3 can be at most 1 after first phase, so we have to consider 1 possibility when odd tasks exist only in I_3. Similarly, for I_4, I_5 and I_6, there are 4, 2, and 3 possibilities respectively when odd tasks exists only in I_4, or only in I_5 or only in I_6. So, when odd tasks are from exactly one of the five subintervals after first phase and four other subintervals have no odd task, such odd tasks are declared as residue tasks. So, when there are residue tasks from

[5]The algorithm OddAssign is again given in Figure 6 from Figure 2 for better readability

Algorithm OddAssign(Odd tasks in subintervals I_2-I_6):
1. **while** both I_2 and I_4 has at least one task
2. Assign $\tau_i \in I_2$ and $\tau_j \in I_4$ to one processor
3. **while** both I_2 and I_5 has at least one task
4. Assign $\tau_i \in I_2$ and $\tau_j \in I_5$ to one processor
5 **while** I_3 has one task and I_6 has two tasks
6. Assign $\tau_i \in I_3$, $\tau_j \in I_6$ and $\tau_k \in I_6$ to one processor
7. **while** ((I_4 has one task and I_5 has two tasks)
8. **or** (I_4 has two tasks and I_5 has one task))
9. **if** (I_4 has one task and I_5 has two tasks) **then**
10. Assign $\tau_i \in I_4$, $\tau_j \in I_5$ and $\tau_k \in I_5$ to one processor
11. **else**
12. Assign $\tau_i \in I_4$, $\tau_j \in I_4$ and $\tau_k \in I_5$ to one processor
13. **end if**
14. **while** I_4 has two tasks and I_6 has one task
15. Assign $\tau_i \in I_4$, $\tau_j \in I_4$ and $\tau_k \in I_6$ to one processor
16. **while** each I_3, I_5 and I_6 has one task
17. Assign $\tau_i \in I_3$, $\tau_j \in I_5$ and $\tau_k \in I_6$ in one processor

Figure 6: Assignment of odd tasks in I_2-I_6

exactly one of the subintervals I_2-I_6, the possibility of residue task in third phase is one of the (2+1+4+2+3=) 12 different possibilities. Each of the 12 possibilities are given in each row of Table 3. The columns in Table 3 represent the number of residue tasks in each of the utilization subinterval I_2–I_6. Observe that, exactly one of the columns in each row of Table 3 is nonzero (since odd tasks are in exactly one subinterval). The minimum and maximum number of residue tasks in exactly one subinterval is also considered in this case. Therefore, after second phase if residue tasks exists in exactly one subinterval I_2–I_6, the possibility is one of these 12 possibilities given in Table 3. Each of the 12 possibilities of residue task has to be considered in third phase of task assignment.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
1	0	0	0	0
2	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	2	0	0
0	0	3	0	0
0	0	4	0	0
0	0	0	1	0
0	0	0	2	0
0	0	0	0	1
0	0	0	0	2
0	0	0	0	3

Table 3: Total 12 possible residue tasks from exactly one subinterval of I_2-I_6

B.2 Case 2: Odd tasks exactly in two subintervals of I_2-I_6

When odd tasks exists in exactly two subuintervals I_2–I_6, there are total ${}^5C_2 = 10$ different ways to consider odd tasks from exactly two subintervals of I_2-I_6. For each of the 10 different ways, if some odd tasks in exactly two of the subintervals I_2-I_6 remain unassigned after second phase, then we consider it as a new possibility of residue tasks. However, if odd tasks remain unassigned in at most one subinterval after second phase, the possibility is same as one of the possibilities already considered in Section B.1. Now we consider each of the 10 different ways to consider odd tasks from any two subintervals I_2-I_6 to assign to processors during second phase and we determine the number of odd tasks in the two subintervals not assigned after second phase.

(1) **I_2 and I_3:** None of the loops of the algorithm OddAssign in Figure 6 considers assigning odd tasks from I_2 and I_3 to processor. Remember that after first phase there are at most 2 tasks in I_2 and at most 1 task in I_3. When odd tasks exists in both subintervals I_2 and I_3 after second phase, there are $(2 \times 1=)$ 2 different possibilities of having residue tasks. First possibility is, one task from I_2 and one from I_3. Second possibility is, two tasks from I_2 and one task from I_3. These two possibilities of odd tasks from these two subintervals I_2 and I_3 would be considered as new possibilities for residue tasks. These two possibilities shown in Table 4.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
1	1	0	0	0
2	1	0	0	0

Table 4: Total 2 possible residue tasks from two subinterval I_2 and I_3

(2) **I_2 and I_4:** The second phase assigns odd tasks from I_2 and I_4 in the *first* while loop using OddAssign in Figure 6. The loop terminates when at least one of the subintervals I_2 or I_4 has *no* tasks. So, after second phase, odd tasks could exist in at most one of the subintervals, either in I_2 or in I_4, but not in both. These unassigned odd tasks in one of the subintervals are declared as residue tasks. However, this possibility of residue task in at most one subinterval is already considered in Section B.1. So, no new possibility of residue tasks is found if odd tasks exists only in I_2 and I_4.

(3) **I_2 and I_5:** The second phase assigns odd tasks from I_2 and I_5 in the *second* while loop using OddAssign in Figure 6. The loop terminates when at least one of the subintervals I_2 or I_5 has *no* odd tasks. So, after second phase, odd tasks could exist in at most one of the subintervals, either in I_2 or in I_5, but not in both. These unassigned odd tasks in one of the subintervals are declared as residue tasks. However, this possibility of residue task in at most one subinterval is already considered in Section B.1. So, no new possibility of residue tasks is found if odd tasks exists only in I_2 and I_5.

(4) **I_2 and I_6:** None of the loops of the algorithm OddAssign in Figure 6 assign odd tasks from both I_2 and I_6 to processors. After first phase, at most 2 odd tasks from I_2 and at most 3 tasks from I_6 may remain unassigned. When odd tasks exists in both subintervals I_2 and I_6, there are total $(2 \times 3=)$ 6 new possibilities of residue tasks. These 6 new possibilities of odd tasks from these two subintervals I_2 and I_6 would be considered as new possibilities of residue tasks. The six possibilities are shown in Table 5.

(5) **I_3 and I_4:** None of the loops of the algorithm OddAssign in Figure 6 assigns odd tasks from I_3 and I_4 to processors. After first phase, at most 1 task from I_3, and at most 4 tasks from I_4 may remain unassigned. When odd tasks exists in both subintervals I_3 and I_4, there are total $(1 \times 4=)$ 4 new possibilities of residue tasks. These 4 new possibilities of odd tasks from these two subintervals I_3 and I_4 would be considered as new possibilities of residue tasks. The four possibilities are shown in Table 6.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
1	0	0	0	1
1	0	0	0	2
1	0	0	0	3
2	0	0	0	1
2	0	0	0	2
2	0	0	0	3

Table 5: Total 6 possible residue tasks from two subintervals I_2 and I_6

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	1	1	0	0
0	1	2	0	0
0	1	3	0	0
0	1	4	0	0

Table 6: Total 4 possible residue tasks from two subintervals I_3 and I_4

(6) **I_3 and I_5:** None of the loops of the algorithm OddAssign in Figure 6 assigns odd tasks from I_3 and I_5 to processor. After first phase, at most 1 task from I_3, and at most 2 tasks from I_5 may remain unassigned. When odd tasks exists in both subintervals I_3 and I_5, there are $(1 \times 2=)$ 2 new possibilities of residue tasks. These 2 new possibilities of odd tasks from these two subintervals would be considered as new possibilities of residue tasks. The two possibilities are shown in Table 7.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	1	0	1	0
0	1	0	2	0

Table 7: Total 2 possible of residue tasks from two subintervals I_3 and I_5

(7) **I_3 and I_6**: The second phase assigns odd tasks from I_3 and I_6 in the *third* while loop using OddAssign in Figure 6. The loop terminates when, either (i) at least one of the subintervals I_3 or I_6 has no tasks, or (ii) there is exactly one task in each subinterval I_3 and I_6. When (i) is true, odd tasks could exist in at most one of the subintervals, either in I_3 or in I_6, but not in both after second phase. These unassigned tasks in exactly one of the subintervals are declared as residue tasks. However, these possibility of residue task from at most one subinterval is already considered in Section B.1. When (ii) is true, exactly one task could exist in each of the subintervals I_3 and I_6 after second phase. These unassigned tasks in exactly two of the subintervals I_3 and I_6 are declared as a new possibility of residue tasks and this possibility shown in Table 8.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	1	0	0	1

Table 8: Total 1 possible of residue tasks from two subintervals I_3 and I_6

(8) **I_4 and I_5:** The second phase assigns odd tasks from I_4 and I_5 in the *forth* while loop using OddAssign in Figure. 6. The loop terminates when, either (i) at least one of the subintervals I_4 or I_5 has no unassigned task , or (ii) there is exactly one task in each subinterval I_4 and I_5. When (i) is true, odd tasks could exist in at most one of the subintervals, either in I_4 or in I_5, but not in both after second phase. These unassigned tasks in one of the subintervals is declared as residue tasks. However, these possibility of residue task from at most one subinterval is already considered in Section B.1. When (ii) is true, exactly one odd task could exist in each of the subintervals I_4 and I_5 after second phase. These unassigned tasks in exactly two of the subintervals I_4 and I_5 are declared as new possibility of residue tasks and this possibility shown in Table 9.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	0	1	1	0

Table 9: Total 1 possible of residue tasks from two subintervals I_4 and I_5

(9) **I_4 and I_6:** The second phase assigns odd tasks from I_4 and I_6 in the *fifth* while loop using OddAssign in Figure 6. The loop terminates when, either (i) at least one of the subintervals I_4 or I_6 has no unassigned task , or (ii) there is 1 odd task in I_4 and there is 1–3 tasks in I_6. When (i) is true, odd tasks could exist in at most one of the subintervals, either in I_4 or in I_6, but not in both after second phase. These unassigned tasks in one of the subintervals is declared as residue tasks. However, these possibility of residue task from exactly one subinterval is already considered in Section B.1. When (ii) is true, one odd task exist in subintervals I_4 and one to three odd tasks exist in I_6 after second phase. These unassigned tasks in two of the subintervals are declared as residue tasks and these $(1 \times 3=)$ 3 possibilities of residue tasks are shown in Table 10.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	0	1	0	1
0	0	1	0	2
0	0	1	0	3

Table 10: Total 3 possible of residue tasks from two subintervals I_4 and I_6

(10) **I_5 and I_6:** None of the loops of the algorithm OddAssign in Figure 6 assigns odd tasks from I_5 and I_6 to processors. After first phase, at most 2 tasks in I_5 and at most 3 tasks in I_6 may remain unassigned. When odd tasks exists in both subintervals I_5 and I_6, there are total $(2 \times 3=)$ 6 new possibilities of residue tasks. These 6 new possibilities of odd tasks from these two subintervals I_5 and I_6 are considered as new possibilities residue tasks. The six possibilities are shown in Table 11.

All the 10 ways to select two of the subintervals I_2–I_6, in which odd tasks may exist after first phase, are considered. After second phase when odd tasks still remain unassigned in exactly two subintervals, there are total (2+6+4+2+1+1+3+6)=25 possibilities. These unassigned odd tasks after second phase are called residue tasks. If residue tasks exist in exactly two subintervals, the possibility of is one of the 25 possibilities for a task set. Each of the 25 possibilities has to be considered in third phase of task assignment.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	0	0	1	1
0	0	0	1	2
0	0	0	1	3
0	0	0	2	1
0	0	0	2	2
0	0	0	2	3

Table 11: Total 6 possibilities of residue tasks from two subintervals I_5 and I_6

B.3 Case 3: Odd tasks exactly in three subintervals of I_2-I_6

When odd tasks exists in exactly three subuintervals I_2–I_6, there are total ${}^5C_2 = 10$ different ways to consider odd tasks from exactly three subintervals of I_2-I_6. For each of the 10 different ways, if some odd tasks in exactly three of the subintervals I_2-I_6 remain unassigned after second phase, then we consider it as a new possibility of residue tasks. However, if odd tasks remain unassigned in at most two subintervals after second phase, the possibility is same as one of the possibilities already considered in Section B.1 or Section B.2. Now we consider each of the 10 different ways to consider odd tasks from any two subintervals I_2-I_6 to assign to processors during second phase and we determine the number of odd tasks in the three subintervals not assigned after second phase.

(1) **I_2, I_3 and I_4**: When odd tasks exist after first phase in subintervals I_2, I_3 and I_4, the *first* while loop of OddAssign in Figure 6 would assign tasks from I_2 and I_4 to processor during second phase. When the loop terminates, at least one of the subintervals I_2 or I_4 has no task. Since at least one of the subintervals has no task, the unassigned odd tasks (if any) from any of the two subintervals I_2 or I_4 (but not both), and odd tasks from I_3 is necessarily same as considering odd tasks in at most two subintervals that is already considered in Section B.1 or Section B.2. So, after second phase no new possibility of residue task is found if odd task exist in these three subintervals I_2, I_3 and I_4 after first phase.

(2) **I_2, I_3 and I_5:** When odd tasks exist after first phase in subintervals I_2, I_3 and I_5, the *second* while loop of OddAssign in Figure 6 during second phase would assign tasks from I_2 and I_5 to processor. When the loop terminates, at least one of the subintervals I_2 or I_5 has no task. Since at least one of the subintervals has no task, the unassigned odd tasks (if any) from any of the subintervals I_2 or I_5 (but not both), and odd tasks from I_3 is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2. So, after second phase no new possibility of residue task is found if odd task exist in these three subintervals I_2, I_3 and I_5 after first phase.

(3) **I_2, I_3 and I_6:** When odd tasks exist after first phase in subintervals I_2, I_3 and I_6, the *third* while loop of OddAssign in Figure 6 would assign tasks from I_3 and I_6 to processor during second phase. When the loop terminates, either (i) at least one of the subintervals I_3 or I_6 has no task, or (ii) exactly one task exist in each subinterval I_3 and I_6. When (i) is true, the unassigned odd tasks (if any) from any of the two subintervals I_3 or I_6 (not from both), and odd tasks from I_2 is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2. When (ii) is true, that is exactly 1 task exist in each subinterval I_3 and I_6, and there are at most 2 tasks in I_2. There is no loop of OddAssign in Figure 6 that assign these odd tasks from these three subintervals. So, there are $(1 \times 2 =)$ 2 new possibilities of residue tasks that might exist in these three subintervals I_2, I_3 and I_6 as shown in Table 12.

(4) **I_2, I_4 and I_5:** When odd tasks exist after first phase in subintervals I_2, I_4 and I_5, the *first* while loop of OddAssign in Figure 6 would assign tasks from I_2 and I_4 to processor

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
1	1	0	0	1
2	1	0	0	1

Table 12: Total 2 possible residue tasks from three subintervals I_2, I_3 and I_6

during second phase. When the loop terminates, at least one of the subintervals I_2 or I_4 has no task. Since at least one of the subintervals has no task, the unassigned odd tasks (if any) from any of the subintervals I_2 or I_4 (but not in both), and odd tasks from I_5 is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2.So, after second phase no new possibility of residue task is found if odd task exist in these three subintervals I_2, I_4 and I_5 after first phase.

(5) **I_2, I_4 and I_6:** When odd tasks exist after first phase in subintervals I_2, I_4 and I_6, the *first* while loop of OddAssign in Figure 6 would assign tasks from I_2 and I_4 to processor during second phase. When the loop terminates, at least one of the subintervals I_2 or I_4 has no task. Since at least one of the subintervals has no task, the unassigned odd tasks (if any) from any of the subintervals I_2 or I_4 (but not in both), and odd tasks from I_6 is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2. So, after second phase no new possibility of residue task is found if odd task exist in these three subintervals I_2, I_4 and I_6 after first phase.

(6) **I_2, I_5 and I_6:** When odd tasks exist after first phase in subintervals I_2, I_5 and I_6, the *second* while loop of OddAssign in Figure 6 during second phase would assign tasks from I_2 and I_5 to processor. When the loop terminates, at least one of the subintervals I_2 or I_5 has no task. Since at least one of the subintervals has no task, the unassigned odd tasks (if any) from any of the subintervals I_2 or I_5 (but not in both), and odd tasks from I_6 is necessarily same as considering odd tasks from at most two subintervals already considered in Section B.1 or Section B.2. So, after second phase no new possibility of residue task is found if odd task exist in these three subintervals I_2, I_5 and I_6 after first phase.

(7) **I_3, I_4 and I_5:** When odd tasks exist after first phase in subintervals I_3, I_4 and I_5, the *fourth* while loop of OddAssign in Figure 6 would assign tasks from I_4 and I_5 to processor during second phase. When the loop terminates, either (i) at least one of the subintervals I_4 or I_5 has no task, or (ii) exactly one task exist in each subinterval I_4 and I_5. When (i) is true, the unassigned odd tasks (if any) from any of the subintervals I_4 or I_5 (not from both), and odd tasks from I_3 is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2. So, no new possibility of residue task is found when (i) is true. When (ii) is true, that is exactly one task exist in each I_4 and I_5, and there could be at most 1 odd task in I_3, there is no loop in OddAssign that assign any of these odd tasks from these three subintervals. So, there is one new possibility of residue tasks that might exist in these three subintervals I_3, I_4 and I_5 as shown in Table 13.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	1	1	1	0

Table 13: Total 1 possible residue tasks from three subintervals I_3, I_4 and I_5

(8) **I_3, I_4 and I_6:** When odd tasks exist after first phase in subintervals I_3, I_4 and I_6, the *third* while loop of OddAssign in Figure 6 would assign tasks from I_3 and I_6 to processor during second phase. When the loop terminates, either (i) at least one of the subintervals I_3 or I_6 has no task, or (ii) exactly one task exist in each subinterval I_3 and I_6. When (i) is true, the

unassigned odd tasks (if any) from any of the subintervals I_3 or I_6 (not from both), and odd tasks from I_4 is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2. So, no new possibility of residue task is found. When (ii) is true, that is exactly one task exist in each I_3 and I_6, and there could be at most 1-4 odd tasks in I_4. Next, the *fifth* loop in OddAssign would assign tasks from I_4 and I_6 to processor during second phase. When the fifth loop terminates, either (iii) at least one of the subintervals I_4 or I_6 has no task, or (iv) exactly one task exist in each subinterval I_4 and I_6.

When (ii) and (iii) are true, the unassigned odd tasks (if any) from any of the subintervals I_4 or I_6 (not from both), and the one odd task from I_3 is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2. So, no new possibility of residue task is found.

When (ii) and (iv) are true, that is exactly one task exist in each I_4 and I_6, and there could be at most 1 task in I_3. There is no loop of OddAssign that assign such odd tasks from these three subintervals I_3, I_4 and I_6. So, there is one possibility of residue tasks that might exist in these three subintervals as shown in Table 13.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	1	1	0	1

Table 14: Total 1 possible residue tasks from three subintervals I_3, I_4 and I_6

(9) **I_3, I_5 and I_6:** When odd tasks exist after first phase in subintervals I_3, I_5 and I_6, the *sixth* while loop of OddAssign in Figure 6 would assign tasks from I_3, I_5 and I_6 to processor during second phase. When the loop terminates, at least one of the subintervals I_3, I_5, or I_6 has no task. Since at least one of the subintervals has no task after the first loop terminates, the unassigned odd tasks (if any) from any of the subintervals I_3, I_5 or I_6 (but not all three) is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2. So, after second phase no new possibility of residue task is found if odd task exist in these three subintervals I_3, I_5 and I_6 after first phase.

(10) **I_4, I_5 and I_6:** When odd tasks exist after first phase in subintervals I_4, I_5 and I_6, the *fourth* while loop of OddAssign in Figure 6 would assign tasks from I_4 and I_5 to processor during second phase. When the loop terminates, either (i) at least one of the subintervals I_4 or I_5 has no task, or (ii) exactly one task exist in each subinterval I_4 and in I_5. When (i) is true, the unassigned odd tasks (if any) from any of the subintervals I_4 or I_5 (not from both), and odd tasks from I_6 is necessarily same as considering odd tasks from at most two subintervals that is already considered in Section B.1 or Section B.2. So, no new possibility of residue task is found. When (ii) is true, then there is exactly one task exist in each I_4 and I_5, and there are at most 3 tasks in I_6. There is no loop of OddAssign that assign tasks from these three subintervals. So, there are $(1 \times 3 =)$ 3 new possibilities of residue tasks that might exist in these three subintervals as shown in Table 15.

$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
0	0	1	1	1
0	0	1	1	2
0	0	1	1	3

Table 15: Total 3 possible residue tasks from three subintervals I_4, I_5 and I_6

All the 10 ways to select three of the subintervals I_2–I_6, in which odd tasks may exist after first phase, are considered. After second phase when odd tasks still remain unassigned in exactly three subintervals, the possibility is one of the (2+1+1+3)=7 possibilities of residue tasks. These unassigned odd tasks after second phase are called residue tasks. If residue tasks exist in exactly three subintervals, the possibility of is one of the 7 possibilities for any task set. Each of the 7 possibilities has to be considered in third phase of task assignment.

Next we would consider odd task from exactly four subintervals. We find that, the number of possibilities of having residue tasks in exactly four subintervals is zero.

B.4 Case 4: Odd tasks exactly in four subintervals of I_2-I_6

When odd tasks exists in exactly four subuintervals I_2–I_6, there are total ${}^5C_4 = 5$ different ways to consider odd tasks from exactly four subintervals of I_2-I_6. For each of the 5 different ways, if some odd tasks in exactly five of the subintervals I_2-I_6 remain unassigned after second phase, then we consider it as a new possibility of residue tasks. However, if odd tasks remain unassigned in at most three subinterval after second phase, the possibility is same as one of the possibilities already considered in Section B.1 or Section B.2 or or Section B.3. Now we consider each of the 5 different ways to consider odd tasks from any four subintervals I_2-I_6 to assign to processors during second phase and we determine the number of tasks in the four subintervals not assigned after second phase.

(1) **I_2, I_3, I_4 and I_5:** When odd tasks exist after first phase in all four subintervals I_2, I_3, I_4 and I_5, the *first* while loop of OddAssign in Figure 6 would assign tasks from I_2 and I_4 to processor during second phase. When the loop terminates, at least one of the subintervals I_2 or I_4 has no task. Since at least one of the subintervals has no task when the first loop terminates, the unassigned odd tasks (if any) from any of the subintervals I_2, I_3, I_4 and I_5 is necessarily same as considering odd tasks from at most three subintervals as already considered in Section B.1-B.3. So, after second phase no new possibility of residue task is found if odd task exist in these four subintervals I_2, I_3, I_4 and I_5 after first phase.

So, no new possibility of residue task is found.

I_2, I_3, I_4 and I_6: When odd tasks exist after first phase in all four subintervals I_2, I_3, I_4 and I_6, the *first* while loop in OddAssign in Figure 6 would assign tasks from I_2 and I_4 to processor during second phase. When the loop terminates, at least one of the subintervals I_2 or I_4 has no task. Since at least one of the subintervals has no task after the first loop terminates, the unassigned odd tasks (if any) from any of the subintervals I_2,I_3,I_4 and I_6 is necessarily same as considering odd tasks from at most three subintervals as already considered in Section B.1-B.3. So, after second phase no new possibility of residue task is found if odd task exist in these four subintervals I_2, I_3, I_4 and I_6 after first phase.

I_2, I_4, I_5 and I_6: When odd tasks exist after first phase in all four subintervals I_2, I_4, I_5 and I_6, the *first* while loop in OddAssign in Figure 6 would assign tasks from I_2 and I_4 to processor during second phase. When the loop terminates, at least one of the subintervals I_2 or I_4 has no task. Since at least one of the subintervals has no task after the first loop terminates, the unassigned odd tasks (if any) from any of the subintervals I_2,I_4,I_5 and I_6 is necessarily same as considering odd tasks from at most three subintervals as as already considered in Section B.1-B.3. So, after second phase no new possibility of residue task is found if odd task exist in these four subintervals I_2, I_4, I_5 and I_6 after first phase.

I_2, I_3, I_5 and I_6: When odd tasks exist after first phase in all four subintervals I_2, I_3, I_5 and I_6, the *second* while loop in OddAssign in Figure 6 would assign tasks from I_2 and I_5 to processor during second phase. When the loop terminates, at least one of the subintervals I_2 or I_5 has no task. Since at least one of the subintervals has no task after the first loop terminates, the unassigned odd tasks (if any) from any of the subintervals I_2,I_3,I_5 and I_6 is necessarily same as considering odd tasks from only at most three subintervals as already considered in Section

B.1, Section B.2 and Section B.3. So, after second phase no new possibility of residue task is found if odd task exist in these four subintervals I_2, I_3, I_5 and I_6 after first phase.

I_3, I_4, I_5 and I_6: When odd tasks exist after first phase in all four subintervals I_3, I_4, I_5 and I_6, at least the *sixth* while loop in OddAssign in Figure 6 would assign tasks from I_3, I_5 and I_6 to processor during second phase. When the loop terminates, at least one of the subintervals I_3, I_5 or I_6 has no task. Since at least one of the subintervals has no task after the first loop terminates, the unassigned odd tasks (if any) from any of the subintervals I_3,I_4,I_5 and I_6 is necessarily same as considering odd tasks from at most three subintervals as as already considered in Section B.1-B.3. So, after second phase no new possibility of residue task is found if odd task exist in these four subintervals I_3, I_4, I_5 and I_6 after first phase.

All the 5 ways to select four of the subintervals I_2–I_6, in which odd tasks may exist after first phase, are considered. When odd tasks exist in exactly four of the five subintervals I_2-I_6 after first task assignment phase, no new possibility of residue task is found in exactly four subintervals since second phase can assign all the odd tasks from at least one of the four subintervals such that residue tasks can exist in at most three subintervals after second phase.

B.5 Case 5: Odd tasks exactly in five subintervals of I_2-I_6

It is easy to see that, there would never be residue tasks in all five subintervals I_2-I_6 since residue tasks from four subintervals are never possible.

In summary, after second phase if residue tasks exist in exactly one of the subintervals of I_2-I_6, there are total 12 possibilities as found in Section B.1. If residue tasks exist in exactly two of the subintervals of I_2-I_6, there are total 25 possibilities as found in Section B.2. If residue tasks exist in exactly three of the subintervals of I_2-I_6, there are total 7 possibilities as found in Section B.3. There is no possibility to have residue task exactly in four or five subintervals of I_2-I_6. So, when residue tasks exists in I_2-I_6, there are total (12+25+7=) 44 possibilities after second phase of task assignment. It is proved that, *residue tasks in subintervals* I_2–I_6 *that need to be handled in the third phase exist as one of the 44 different possibilities for any task set after second phase.*

We write a C program to generate all the 44 possibilities of residue tasks in I_2–I_6 for schedulability analysis. The pseudocode of the program is given in Subsection B.6 for the interested readers to find all the 44 possibilities of residue tasks.

B.6 Pseudocode to Generate All 44 Possibilities of Residue Tasks

For any task set, after second phase of task assignment in `IBPS`*if some residue task exists, then the residue tasks in subintervals* I_2–I_6 *that need to be handled in the third phase exist as one of the 44 different possibilities.* We present the pseudocode `GenResidue-44` that is used to find the number of residue tasks in each of the subinterval I_2-I_6 for all the 44 possibilities.

The value of each of the five loop variables in line 2-6 represents the number of odd tasks after first phase in each iteration. In line 7-11 of `GenResidue-44`, the number of odd tasks in each subinterval I_2-I_6 is determined for one of the 360 possibilities after first phase. For each such possibility, the algorithm OddAssign is called at line 12. If algorithm OddAssign could not assign all the odd tasks in I_2-I_6 for such a possibility of odd tasks, and the remaining such unassigned odd tasks is not considered as a possibility of residue tasks yet (checked at line 13), then these unassigned odd tasks are declared as one new possibility of residue tasks that need to be considered in the third phase. We find that, the condition at line 13 is true for exactly 44 times. Each time the condition at line 13 is true, the content of each R[i] is output to represent the number of residue tasks in the subinterval I_i for $i = 2, 3 \ldots 6$ as one of the 44 possibilities. All the 44 possibilities of residue tasks is given in Table 16. The rows of the Table 16 are ordered in such a way that is used as the first six columns for Table 2 in Section 7. Notice that, all the rows of the Table 16 are formally determined in Sections B.1-B.5.

Pseudocode `GenResidue-44`

```
1.  Initialize Array R[2..6]={0,0,0,0,0} to represent no odd task in I_2-I_6
2.  For i_2=0 to 2
3.    For i_3=0 to 1
4.      For i_4=0 to 4
5.        For i_5=0 to 2
6.          For i_6=0 to 3
7.              R[0]=i_1
8.              R[1]=i_2
9.              R[2]=i_3
10.             R[3]=i_4
11.             R[4]=i_5
                comment: we generate R[i] number of task in each I_i
12.             Call OddAssign(I_2-I_6)
13.             if some R[i] is not zero and this possibility is not output before then
14.               Print all R[i] for i = 2, 3 ... 6 as one new possibility of residue task after second phase.
```

Figure 7: Pseudocode to generate all 44 different possibilities of residue task in I_2-I_6

No.	$\#I_2$	$\#I_3$	$\#I_4$	$\#I_5$	$\#I_6$
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	1	0	0
4	0	1	0	0	0
5	0	0	0	0	2
6	0	0	0	1	1
7	0	0	0	2	0
8	0	0	1	0	1
9	0	0	1	1	0
10	0	1	0	0	1
11	1	0	0	0	0
12	0	0	0	0	3
13	0	0	2	0	0
14	0	1	0	1	0
15	0	0	0	1	2
16	0	0	0	2	1
17	0	1	1	0	0
18	0	0	1	0	2
19	0	0	1	1	1
20	1	0	0	0	1
21	0	0	0	1	3
22	0	1	0	2	0
23	0	0	0	2	2
24	0	0	1	0	3
25	0	0	3	0	0
26	0	1	1	0	1
27	0	1	1	1	0
28	0	0	1	1	2
29	1	1	0	0	0
30	0	1	2	0	0
31	1	0	0	0	2
32	0	0	0	2	3
33	0	0	1	1	3
34	1	0	0	0	3
35	0	0	4	0	0
36	1	1	0	0	1
37	2	0	0	0	0
38	2	0	0	0	1
39	2	1	0	0	0
40	0	1	3	0	0
41	2	0	0	0	2
42	2	1	0	0	1
43	2	0	0	0	3
44	0	1	4	0	0

Table 16: Total 44 possibilities of residue tasks from subintervals I_2-I_6

C Dispatcher

In this appendix, the dispatcher of IBPS is presented. The *run time dispatcher* of each processor is very simple: a dispatcher that considers task offset for Rate-Monotonic scheduling is used for runtime dispatching of tasks for execution. The high-level overview of the runtime dispatcher for each processor is given in Figure 8. Whenever a non-split task arrives to a processor Θ_l it is stored in ready queue $Q[l]$. When a split task τ_i arrives to processor Θ_l, then the first subtask is stored is ready queue $Q[l]$ while the second subtask is stored in ready queue $Q[l+1]$. Each processor executes the highest priority task from the ready queue.

Algorithm DISPATCHER

1. Let each processor Θ_l has a local queue $Q[l]$ where ready (or preempted tasks) are stored
2. **for** processors Θ_l in $l \in \{1 \dots m\}$
3. **When** the system starts **do**
4. Initialize the ready queue $Q[l]$ as empty
5. **end do**
6. *Comment: assume a split task arrives to a processor to which the first split subtask is assigned*
7. **When** any task τ_i arrives on processor Θ_l **do**
8. **if** task τ_i is a split task **then**
9. Store in $Q[l]$ the subtask $\tau_i{}' = (\frac{C_i}{2}, T_i, 0)$
10. Store in $Q[l+1]$ the subtask $\tau_i'' = (\frac{C_i}{2}, T_i, \frac{C_i}{2})$
11. **else**
12. Store in $Q[l]$ task $\tau_i \ = (C_i, T_i, 0)$
13. **end if**
14. **end do**
15. Execute task from $Q[l]$ in RM priority order considering offset of each task.

Figure 8: High level overview of the Run-Time Dispatcher of IBPS