A PTAS for assigning sporadic tasks on two-type heterogeneous multiprocessors

Gurulingesh Raravi
Vincent Nelis

HURRAY-TR-120505
Version:
Date: 5/17/2012
Abstract

Consider the problem of determining a task-to-processor assignment for a given collection of implicit-deadline sporadic tasks upon a multiprocessor platform in which there are two distinct kinds of processors. We propose a polynomial-time approximation scheme (PTAS) for this problem. It offers the following guarantee: for a given task set and a given platform, if there exists a feasible task-to-processor assignment, then given an input parameter, 'e', our PTAS succeeds, in polynomial time, in finding such a feasible task-to-processor assignment on a platform in which each processor is 1+3e times faster. In the simulations, our PTAS outperforms the state-of-the-art PTAS and also for the vast majority of task sets, it requires significantly smaller processor speedup than (its upper bound of) 1+3e for successfully determining a feasible task-to-processor assignment.
A PTAS for assigning sporadic tasks on two-type heterogeneous multiprocessors

Gurulingesh Raravi and Vincent Nélis
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
Email: ghri@isep.ipp.pt, nelis@isep.ipp.pt

Abstract—Consider the problem of determining a task-to-processor assignment for a given collection of implicit-deadline sporadic tasks upon a multiprocessor platform in which there are two distinct kinds of processors. We propose a polynomial-time approximation scheme (PTAS) for this problem. It offers the following guarantee: for a given task set and a given platform, if there exists a feasible task-to-processor assignment, then given an input parameter, $\epsilon$, our PTAS succeeds, in polynomial time, in finding such a feasible task-to-processor assignment on a platform in which each processor is $1+3\epsilon$ times faster. In the simulations, our PTAS outperforms the state-of-the-art PTAS [1] and also for the vast majority of task sets, it requires significantly smaller processor speedup than (its upper bound of) $1+3\epsilon$ for successfully determining a feasible task-to-processor assignment.

I. INTRODUCTION

This paper addresses the problem of finding an assignment of tasks to processors (also referred to as partitioning) for a given set of implicit-deadline sporadic tasks (also referred to as Liu and Layland (LL) tasks [2]) on a heterogeneous multiprocessor platform comprising processors of two unrelated types: type-1 and type-2. We refer to such a computing platform as a two-type platform. Our interest in considering such a platform model is motivated by the fact that many chip makers offer chips having two types of processors [3]–[7].

In the partitioning problem, every task must be statically assigned to a processor at design time and all its jobs must execute on that processor at run time. The challenge is to find, at design time, a task-to-processor assignment such that, at run time, an uniprocessor scheduling algorithm running on each processor meets all the deadlines. Scheduling the tasks to meet deadlines on an uniprocessor platform is a well-understood problem. One may use Earliest-Deadline First (EDF) [2], for example. EDF is an optimal scheduling algorithm on uniprocessor systems [2], [8], with the interpretation that it always constructs a schedule in which all the deadlines are met, if such a schedule exists. Therefore, assuming that an optimal scheduling algorithm is used on each processor, the challenging part is to find a partitioning for which there exists a schedule that meets all the deadlines — such a partitioning is said to be a feasible partitioning hereafter. Even in the simpler case of identical multiprocessors, finding a feasible partitioning is strongly NP-Complete [9]. Hence, this result continues to hold for two-type platforms. In this work, we propose a polynomial-time approximation scheme (PTAS), for this problem which outperforms the state-of-the-art PTAS [1].

Definition 1 (PTAS). A PTAS takes an instance of an optimization problem (for which exact solutions are intractable) and a parameter $\epsilon > 0$ and, in polynomial time, produces a solution that is within a factor $f(\epsilon)$ of being optimal where function $f(\epsilon)$ is independent of the problem instance.

Definition 2 (Approximation ratio). An algorithm for solving an optimization problem is said to have an approximation ratio of $A$ if for all instances of the problem, the algorithm produces a solution that is within a factor of $A$ from the optimal value.

Related work. The partitioning problem on heterogeneous multiprocessors has been studied in the past [10]–[14]. In [10]–[12], the authors proposed algorithms for the problem of partitioning LL task sets on heterogeneous multiprocessors with an approximation ratio of 2. All these approaches [10]–[12] focused on generic heterogeneous multiprocessor platforms with two or more processor types. Due to practical relevance, Andersson et al. [13] considered the partitioning problem on two-type platforms and proposed an algorithm, FF-3C, and couple of its variants based on first-fit heuristic. These had the same performance guarantee as the approaches in [10]–[12] (i.e., requiring processors twice as fast, in the worst-case) but can be implemented efficiently and exhibited better average-case performance than those in [11], [12].

In a recent significant development, Wiese et al. [1] proposed a PTAS (referred to as PTAS$_{LP}$ as it uses “Linear Programming”) for partitioning LL task system on limited heterogeneous multiprocessors in which processors are of a relatively small number ($\geq 2$) of distinct types. The PTAS$_{LP}$ provides the following guarantee: if there exists a feasible partitioning of a given task set on a limited heterogeneous multiprocessor platform then the PTAS$_{LP}$ succeeds in partitioning the task set on a platform in which each processor is $\frac{1+\epsilon}{1-\epsilon}$ times faster. This is theoretically a significant result since PTAS$_{LP}$ partitions the task set in polynomial time, to any desired degree of accuracy. However, its practical significance is severely limited as the algorithm has a very high run-time complexity since it “heavily” relies on solving the linear program. Even on a two-type platform, it has a high run-time complexity which makes its implementation highly inefficient (which is confirmed by the simulations). Therefore, we propose a PTAS for two-type platforms which does not rely on solving linear programs and hence offers a significantly better time-complexity than PTAS$_{LP}$.

Contribution and significance of this work. We present a PTAS for the problem of partitioning an LL task set on a two-type platform which offers the following guarantee. If there exists a feasible partitioning of a task set $\tau$ on a two-type
platform $\pi$ then given an $\epsilon > 0$, PTAS succeeds, in polynomial time, in finding a feasible partitioning of $\pi$ on $\pi^{(1+3\epsilon)}$ where $\pi^{(1+3\epsilon)}$ is a two-type platform in which each processor is $1+3\epsilon$ times faster than the corresponding processor in $\pi$.

We believe the significance of this work is as follows. For the problem under consideration, our PTAS has superior performance compared to state-of-the-art, PTAS$_{LP}$. Specifically, compared to PTAS$_{LP}$, our PTAS has (i) a much better runtime complexity and (ii) a competitive approximation ratio. We evaluate the average-case performance of these algorithms with randomly generated task sets. The evaluation is based on (i) the processor speedup the algorithm needs, for a given task set, so as to succeed, compared to an optimal algorithm and (ii) the running time. Overall, our algorithm outperforms PTAS$_{LP}$ by requiring much smaller processor speedup and running faster by orders of magnitude. Also, for the vast majority of task sets, it requires significantly smaller processor speedup than its upper bound of $1+3\epsilon$.

II. System Model

We consider the problem of partitioning a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of $n$ implicit-deadline sporadic tasks (LL tasks) on a two-type heterogeneous multiprocessor platform comprising $m$ processors, of which $m_1$ are of type-1 and $m_2$ are of type-2. Each task $\tau_i$ is characterized by two parameters: a worst-case execution time (WCET) and a period $T_i$. Each task $\tau_i$ releases a (potentially infinite) sequence of jobs, with the first job released at any time during the system execution and subsequent jobs released at least $T_i$ time units apart. Each job released by a task $\tau_i$ has to complete its execution within $T_i$ time units from its release. We assume that an optimal scheduling algorithm such as EDF is used on each processor.

On a two-type platform, the WCET of a task depends on the processor type on which it executes. We denote by $C^1_i$ and $C^2_i$ the WCET of task $\tau_i$ on processors of type-1 and type-2 and we denote by $u_i = \frac{C^1_i}{T_i}$ and $v_i = \frac{C^2_i}{T_i}$ its utilizations on type-1 and type-2 processors, respectively. A task $\tau_i$ that cannot be executed on processors of type-1 (resp., type-2) is modeled by setting $u_i = \infty$ (resp., $v_i = \infty$).

III. An Overview of Our Approach

We now give an overview of our algorithm, PTAS$_{NF}$ (NF stands for “Next-Fit”). Our PTAS takes $\epsilon > 0$ as an input parameter and outputs a feasible partitioning. Let us partition the given task set $\tau$ into two subsets as follows:

$$\tau_{hvy} = \{\tau_i \mid u_i \geq \epsilon \text{ or } v_i \geq \epsilon\}$$

(1)

$$\tau_{lgt} = \tau \setminus \tau_{hvy} = \{\tau_i \mid u_i < \epsilon \text{ and } v_i < \epsilon\}$$

(2)

Intuitively, $\tau_{hvy}$ refers to “heavy” and $\tau_{lgt}$ to “light” tasks. Our PTAS, has the following steps:

Step 1. We first approximate the utilizations, $u_i$ and $v_i$, of every task $\tau_i \in \tau_{hvy}$ to some pre-computed values. The motivation for doing this is twofold: by (i) restricting the number of pre-computed values to a constant, we ensure polynomial complexity for the algorithm and (ii) choosing these values cleverly, we ensure the approximation ratio of the algorithm is bounded. Then, we assign the tasks in $\tau_{hvy}$ to processors using the algorithm $A_{hvy}$ described in Section IV-A. In Section IV-E, we show that after using $A_{hvy}$, the sum of the utilizations of the tasks assigned on processors of type-1 (resp., type-2) does not exceed $(1 + \epsilon) \times m_1$ (resp., $(1 + \epsilon) \times m_2$).

Step 2. Some tasks from $\tau_{hvy}$ (with $u_i \geq \epsilon$ and $v_i < \epsilon$ or $u_i < \epsilon$ and $v_i \geq \epsilon$) may remain unassigned after using $A_{hvy}$. These unassigned tasks form the set, $\tau_{int}$ ("intermediate") tasks. Now, $A_{int}$ (resp., $A_{rd}$) assigns the tasks (i.e., tasks can be split between processors) with $u_i < \epsilon$ and $v_i \geq \epsilon$ (resp., $u_i \geq \epsilon$ and $v_i < \epsilon$) to type-1 (resp., type-2) processors as described in Section V-A. In Section V-B, we show that after using $A_{int}$, the sum of the utilizations of all the tasks assigned so far on processors of type-1 (resp., type-2) still does not exceed $(1 + \epsilon) \times m_1$ (resp., $(1 + \epsilon) \times m_2$).

Step 3. Finally, those tasks from $\tau_{lgt}$ and $\tau_{rd}$ that were assigned fractionally by $A_{int}$ and $A_{rd}$ are assigned integrally using the algorithm $A_{fract}$ described in Section VII-A. In Section VII-B, we show that after using $A_{fract}$, the sum of the utilizations of all the tasks assigned so far on processors of type-1 (resp., type-2) does not exceed $(1 + 3\epsilon) \times m_1$ (resp., $(1 + 3\epsilon) \times m_2$). Hence, we conclude that if $\tau$ has a feasible partitioning on $\pi$ then PTAS$_{NF}$ succeeds in finding such a feasible partitioning of $\tau$ on $\pi^{(1+3\epsilon)}$.

IV. Assigning the Tasks in $\tau_{hvy}$ (Step 1)

In this section, we describe the algorithm, $A_{hvy}$, for integrally assigning (a subset of) the tasks in $\tau_{hvy}$ to processors and also analyze its returned assignment.

A. Description of the algorithm $A_{hvy}$

It consists of three steps described in the next three sections:

Step 1.1. It defines a finite set $S(\epsilon)$ of utilization values, based on the value of the input parameter, $\epsilon$. Then, it computes the "rounded-down utilizations" $u_{i, rd}$ and $v_{i, rd}$ of every task $\tau_i \in \tau$ by rounding down $u_i$ and $v_i$ to one of the quantized values in $S(\epsilon)$. We will denote by $\tau_{hvy}^{rd}$ the set of tasks obtained by rounding down the utilizations of the tasks of $\tau_{hvy}$.

Step 1.2. It uses dynamic programming to determine, in polynomial time, (i) all the subsets of $\tau_{hvy}^{rd}$ that can be partitioned upon $m_1$ processors of type-1 and (ii) all the subsets that can be partitioned upon $m_2$ processors of type-2.

Step 1.3. It exhaustively considers each pair of subsets such that one subset can be partitioned on $m_1$ processors of type-1 and the other subset can be partitioned on $m_2$ processors of type-2. Using the ordered pair of subsets under consideration, it integrally assigns (a subset of the) tasks from $\tau_{hvy}$ to processors (at least those with $u_i \geq \epsilon$ and $v_i \geq \epsilon$).

B. Rounding-down the utilizations of the tasks (Step 1.1)

We compute the set $S(\epsilon)$ of all real numbers $\leq 1$ that are of the form $(1 + \epsilon)^k$, for all integers $k \geq 0$. Then, we compute
the rounded-down utilizations $u_i^{rd}$ and $v_i^{rd}$ of every task $\tau_i \in \tau$ by rounding down each of its utilizations ($u_i$ and $v_i$) to the nearest value present in the set $S(\epsilon)$. If there is no such value in $S(\epsilon)$ (i.e., if $u_i$ or $v_i$ is $< \epsilon$) then the corresponding rounded-down utilization is set to 0. For those tasks whose $u_i$ or $v_i$ is set to $\infty$, we set their rounded-down utilizations to $\infty$ as well. The definition of $S(\epsilon)$ leads to the following property.

**Property 1.** For a task $\tau_i$, if $\epsilon \leq u_i \leq 1$ then there exists $k$ such that $\epsilon (1 + \epsilon)^k \leq u_i < \epsilon (1 + \epsilon)^{k+1}$ and thus

$$u_i^{rd} = \frac{u_i}{\epsilon (1 + \epsilon)^k} \leq \epsilon (1 + \epsilon)^k (1 + \epsilon)^k = (1 + \epsilon)^k$$

(3)

The same holds for $v_i$.

Therefore, if the utilizations of each task is reduced by this maximal factor, it follows that any collection of tasks with their reduced utilizations summing to $\leq 1$ would have their original utilizations summing to $\leq (1 + \epsilon)$.

Let us now determine the number $L$ of distinct values in $S(\epsilon)$. Since only values $\epsilon (1 + \epsilon)^k \leq 1$ are included in $S(\epsilon)$, it holds that $k \log (1 + \epsilon) \leq \log (1/\epsilon)$ and thus, $k \leq \log (1/\epsilon) / \log (1 + \epsilon)$. Then we conclude that $L = \left\lceil \log (1/\epsilon) / \log (1 + \epsilon) \right\rceil + 1$.

For each $\ell$, $0 \leq \ell < L$, we denote by $X_\ell$ (resp., $Y_\ell$) the number of tasks in $T_{hvy}$ with $u_i^{rd}$ (resp., $v_i^{rd}$) equal to $\epsilon (1 + \epsilon)^\ell$ in $S(\epsilon)$. The task set $T_{hvy}$ can thus be represented by $2 \times L$ non-negative integers $X_0, X_1, \ldots, X_{L-1}, Y_0, Y_1, Y_{L-1}$. Note that each $X_\ell$ and each $Y_\ell$ is no greater than $|T_{hvy}|$.

C. Generating the feasible configurations (Step 1.2)

The rounding down of the utilizations described in the previous section ensures that the utilizations of the tasks in $T_{hvy}$ may only take one of the values in $S(\epsilon)$, providing the set $T_{hvy}$. In this section, using dynamic programming, we determine, in polynomial time, all the subsets of $T_{hvy}$ that can be partitioned upon $m_1$ processors of type-1 (resp., $m_2$ processors of type-2). Once all the feasible subsets (also referred to as feasible configurations) are determined, we use this information to assign a subset of tasks from $T_{hvy}$ on type-1 and type-2 processors (described in Section IV-D).

**Definition 3 (feasible configurations).** Consider any $L$-tuple $T = (x_0, x_1, \ldots, x_{L-1})$ where $x_\ell \geq 0, \forall \ell \in [0, L - 1]$, and let $\tau(T)$ denote a task set containing exactly $x_\ell$ tasks $\tau_i$ of utilization $u_i = \epsilon (1 + \epsilon)^\ell$ for each $\ell$. The $L$-tuple $T$ is said to be a feasible configuration on $m_1$ processors of type-1 if and only if there exists a feasible partitioning for the corresponding task set $\tau(T)$ on $m_1$ processors of type-1. Analogously, we define an $L$-tuple $(y_0, y_1, \ldots, y_{L-1})$ with $v_\ell$ values that is a feasible configuration on $m_2$ processors of type-2.

The algorithm $A_{hvy}$ uses the same approach as the one presented in [14] to determine all the configurations $(x_0, x_1, \ldots, x_{L-1})$ of tasks in $T_{hvy}^{rd}$ (resp., $(y_0, y_1, \ldots, y_{L-1})$ of tasks in $T_{hvy}^{rd}$) that are feasible on $m_1$ processors of type-1 (resp., $m_2$ processors of type-2), in which $x_\ell \leq X_\ell \leq |T_{hvy}|$ (resp., $y_\ell \leq Y_\ell \leq |T_{hvy}|$) for each $\ell$, $0 \leq \ell < L$. This approach [14] is summarized below. As there are no more than $\Pi_{\ell=0}^{L-1} (1 + X_\ell) \leq \Pi_{\ell=0}^{L-1} (1 + |T_{hvy}|) = (nL)$ such feasible configurations on type-1 processors (and the same holds for type-2 processors) and since $L$ is a constant for a given value of $\epsilon$, the time to determine all the feasible configurations is polynomial in $n$.

**Summary of the approach in [14]:** It constructs two separate tables: one table each for storing the information about all the configurations on processors of each type. The table for type-1 processors has $m_1$ rows and $\Pi_{\ell=0}^{L-1} (1 + X_\ell)$ columns. Each column corresponds to a different configuration and each cell has a value in $\{\text{yes, no}\}$. A cell in the $i$'th row and $j$'th column is a “yes” if the corresponding configuration is feasible on $i$ processors of type-1. This table is filled row by row starting with the first row. Filling in the first row is straightforward for all the configurations: it is a “yes” if the corresponding configuration, say $(x_0, x_1, \ldots, x_{L-1})$, is feasible on a single processor, i.e., if $\sum_{\ell=0}^{L-1} x_\ell \times \epsilon (1 + \epsilon)^\ell \leq 1$, it is a “no” otherwise. The $i$'th row is filled in by using the entries of the $(i-1)$'th row. Specifically, for the configuration corresponding to the $j$'th column, say $(x_0, x_1, \ldots, x_{L-1})$, the cell at the $i$'th row is a “yes” if and only if there exists two configurations $(x_0', x_1', \ldots, x_{L-1}')$ and $(x_0'', x_1'', \ldots, x_{L-1}'')$ such that

1. $(x_0', x_1', \ldots, x_{L-1}')$ is a feasible configuration on $(i-1)$ processors of type-1;
2. $(x_0'', x_1'', \ldots, x_{L-1}'')$ is a feasible configuration on one processor of type-1; and
3. $x_\ell = x_\ell' + x_\ell''$, for all $0 \leq \ell < L$.

For each cell in the $j$'th row, there are polynomially many possible candidates for the role of $(x_0', x_1', \ldots, x_{L-1}')$; hence, each cell in the $j$'th row can be filled in polynomial time [14]. Analogously, the second table for the configurations on type-2 processors is constructed.

**D. Determining the partitioning (Step 1.3)**

Using the two configuration tables that were constructed in the previous step, we now determine a partitioning for a subset of the heavy tasks. The main idea is as follows. Suppose that the task set $\tau$ can indeed be partitioned on the given platform and let $H_{feas}$ denote (one of) the feasible partitioning. For each $\ell$, $0 \leq \ell < L$, let $x_\ell^{feas}$ denote the number of tasks $\tau_i$ satisfying $\epsilon (1 + \epsilon)^\ell < u_i < \epsilon (1 + \epsilon)^{\ell+1}$ that are assigned to type-1 processors in $H_{feas}$. Since $H_{feas}$ is a feasible partitioning, the configuration $(x_0^{feas}, x_1^{feas}, \ldots, x_{L-1}^{feas})$ must appear in the table constructed (in the previous step) for type-1 processors and the cell at the $m_1$'th row of the corresponding column must contain “yes”. Analogously, the configuration $(y_0^{feas}, y_1^{feas}, \ldots, y_{L-1}^{feas})$ must appear in the table constructed for type-2 processors and the cell at the $m_2$'th row of the corresponding column must contain “yes”. However, since we do not know which of the feasible configurations in our tables correspond to $H_{feas}$, we consider every ordered pair of configurations that are feasible on $m_1$ and $m_2$ processors of type-1 and type-2 respectively. Since there are only polynomially (i.e., $O(nL)$) many distinct feasible configurations in each table, it follows that there are at most polynomially many such ordered pairs of feasible configurations to consider.

For each considered ordered pair of configurations, by assuming that they are the ones corresponding to $H_{feas}$, we
attempt to construct a similar task-to-processor assignment for the tasks in \( \tau_{hvy} \) as that of \( \mathcal{H}_{feas} \). The assignment obtained will be similar to \( \mathcal{H}_{feas} \) in the following sense: although the tasks assigned in both the assignments may not be the same, it holds that (as we show later), the sum of utilizations of the tasks assigned by our algorithm on each processor type does not exceed that of \( \mathcal{H}_{feas} \) by a factor of \( 1 + \epsilon \).

Let \( \{ (x_0, x_1, \ldots, x_{L-1}), (y_0, y_1, \ldots, y_{L-1}) \} \) denote the currently considered ordered pair of feasible configurations on \( n_1 \) and \( n_2 \) processors of type-1 and type-2, respectively. The algorithm \( A_{hvy} \) to determine the corresponding task-to-processor assignment of tasks from \( \tau_{hvy} \) is as follows.

**Step 1.3.1.** For each \( \ell \), \( 0 \leq \ell \leq L - 1 \), \( A_{hvy} \) assigns exactly \( x_\ell \) tasks \( \tau_i \) satisfying \( v_i^{rd} = \epsilon (1 + \epsilon)^\ell \) to type-1 processors. Specifically, for each \( \ell \),

1. If there are fewer than \( x_\ell \) such tasks in \( \tau_{hvy} \), then \( A_{hvy} \) declares failure with respect to this particular ordered pair of feasible configurations, and moves on to the next ordered pair of feasible configurations.
2. If there are exactly \( x_\ell \) such tasks then \( A_{hvy} \) assigns all of them to type-1 processors.
3. If there are more than \( x_\ell \) such tasks, it assigns \( x_\ell \) of them to type-1 processors by favoring those with larger \( v_i \).

**Step 1.3.2.** After assigning tasks to processors of type-1, \( A_{hvy} \) assigns the remaining tasks to processors of type-2 as follows. For each \( \ell \), starting with \( \ell = L - 1 \) and repeatedly decreasing \( \ell \) by one until \( \ell = 0 \),

1. If there are less than \( y_\ell \) unassigned tasks \( \tau_i \) satisfying \( v_i^{rd} = \epsilon (1 + \epsilon)^\ell \) (say, \( n_1 \) tasks), then \( A_{hvy} \) assigns these \( n_1 \) tasks to type-2 processors. Then, \( A_{hvy} \) assigns \( y_\ell - n_1 \) other (unassigned) tasks \( \tau_j \) with smaller utilization on type-2 processors (i.e., \( v_j^{rd} < \epsilon (1 + \epsilon)^\ell \)), by favoring those with larger \( v_j \) and \( v_i \) within these tasks that are favored, those with larger \( u_i \) are favored.
2. If there are exactly \( y_\ell \) unassigned tasks \( \tau_i \) satisfying \( v_i^{rd} = \epsilon (1 + \epsilon)^\ell \) then all of them are assigned to type-2 processors.
3. If there are more than \( y_\ell \) unassigned tasks \( \tau_i \) satisfying both (i) \( v_i^{rd} = \epsilon (1 + \epsilon)^\ell \) and (ii) \( u_i^{rd} > 0 \), then \( A_{hvy} \) declares failure with respect to this particular ordered pair of feasible configurations and moves on to the next ordered pair of feasible configurations.
4. If there are more than \( y_\ell \) unassigned tasks \( \tau_i \) satisfying \( v_i^{rd} = \epsilon (1 + \epsilon)^\ell \) but not more than \( y_\ell \) of these tasks have \( u_i^{rd} > 0 \), then \( A_{hvy} \) assigns \( y_\ell \) of these tasks by favoring those with larger \( u_i \).

**Step 1.3.3.** If any task \( \tau_i \) remains unassigned with both \( u_i^{rd} > 0 \) and \( v_i^{rd} > 0 \), \( A_{hvy} \) declares failure with respect to this particular ordered pair of feasible configurations, and moves on to the next ordered pair of feasible configurations.

If \( A_{hvy} \) did not declare failure in any of the above steps, implying that all the tasks with \( u_i \geq \epsilon \land v_i \geq \epsilon \) are assigned (and may be few other tasks from \( \tau_{hvy} \) with \( u_i \geq \epsilon \land v_i < \epsilon \) or \( u_i < \epsilon \land v_i \geq \epsilon \) then algorithm \( A_{int} \) is called with the ordered pair of feasible configurations under consideration. This algorithm, \( A_{int} \), is presented in Section V.

### E. Assignment analysis

Let \( \mathcal{H}_{hvy} \) denote the assignment of the heavy tasks returned by \( A_{hvy} \). In this section, we show that in \( \mathcal{H}_{hvy} \), the subset of tasks assigned to each processor consume no more than \((1 + \epsilon)\) of the capacity of that processor.

**Definition 4 (The subsets \( \Gamma_{hvy}^1 \) and \( \Gamma_{hvy}^2 \)).** We denote by \( \Gamma_{hvy}^1, \Gamma_{hvy}^2 \subseteq \tau_{hvy} \) the subsets of tasks assigned to the processors of type-1 (resp., type-2) in the assignment \( \mathcal{H}_{hvy} \) returned by the algorithm \( A_{hvy} \).

**Note:** Hereafter, we use the notation \( \tau \) for the subsets of tasks that we explicitly define (like \( \tau_{hvy} \) and \( \tau_{opt} \), for example), \( \Gamma \) for the subsets of tasks returned by the different steps of our PTAS and \( \Phi \) for the subsets of tasks assigned in \( \mathcal{H}_{feas} \).

We know that the ordered pair of feasible configurations \( \{ (x_0^{feas}, x_1^{feas}, \ldots, x_{L-1}^{feas}), (y_0^{feas}, y_1^{feas}, \ldots, y_{L-1}^{feas}) \} \) corresponding to the feasible partitioning \( \mathcal{H}_{feas} \) must be present in the tables constructed in Step 1.2 (in Section IV-C). Therefore, this particular ordered pair of feasible configurations (denoted by \( \mathcal{P}_{feas} \) hereafter) will come to be considered by \( A_{hvy} \).

**Lemma 1.** If \( \mathcal{P}_{feas} \) is the ordered pair of feasible configurations currently under consideration by \( A_{hvy} \), then \( A_{hvy} \) successfully terminates (i.e., without declaring failure) and it holds that every task \( \tau_i \in \Gamma_{hvy}^1 \) can be 1:1 mapped to exactly one task \( \tau_k \) that is assigned to a type-1 processor in \( \mathcal{H}_{feas} \) such that \( u_i \leq (1 + \epsilon)v_k \). An analogous property holds for the tasks in \( \Gamma_{hvy}^2 \) (such that \( v_i \leq (1 + \epsilon)u_k \)).

**Proof:** First, let us focus on the tasks in \( \Gamma_{hvy}^1 \). In Step 1.3.1, for each \( \ell \in [0, L - 1) \), it is straightforward (from the fact that we consider the ordered pair \( \mathcal{P}_{feas} \)) to see that \( A_{hvy} \) successfully assigns exactly \( x_\ell^{feas} \) tasks \( \tau_i \) satisfying \( \epsilon (1 + \epsilon)^\ell \leq u_i < \epsilon (1 + \epsilon)^{\ell+1} \) to type-1 processors (through either case 1.3.1.2 or 1.3.1.3). While these may not be the same tasks as those that are assigned to these processors in \( \mathcal{H}_{feas} \), the utilization of each task does not exceed that of the corresponding task assigned in \( \mathcal{H}_{feas} \) by more than a factor of \((1 + \epsilon)\). Hence the lemma holds for the heavy tasks in \( \Gamma_{hvy}^1 \).

Now, let us focus on Step 1.3.2, i.e., on the tasks in \( \Gamma_{hvy}^2 \). If \( A_{hvy} \) terminates without declaring failure then it means that for each \( \ell \in [0, L - 1) \), \( A_{hvy} \) went through either case 1.3.2.1, 1.3.2.2 or 1.3.2.4 and it is trivial to see that the lemma holds for all these cases. Indeed, for each task \( \tau_i \) with \( \epsilon (1 + \epsilon)^\ell \leq u_i < \epsilon (1 + \epsilon)^{\ell+1} \) that is assigned to processors of type-2 through one of these cases, there is a task, say \( \tau_k \), also with \( \epsilon (1 + \epsilon)^\ell \leq v_k < \epsilon (1 + \epsilon)^{\ell+1} \) which is also assigned to processors of type-2 in \( \mathcal{H}_{feas} \) (since we consider the ordered pair \( \mathcal{P}_{feas} \)).

Since we have shown that the lemma holds as long as \( A_{hvy} \) does not declare failure, we now show that \( A_{hvy} \) cannot fail while considering the ordered pair \( \mathcal{P}_{feas} \) of feasible configurations. For a failure to occur, it is necessary for \( A_{hvy} \) to go through case 1.3.2.3, i.e., there must be some \( \ell \in [0, L - 1) \) such that there are strictly more than \( y_\ell^{feas} \) tasks \( \tau_i \) yet unassigned, that satisfy both \( u_i^{rd} = \epsilon (1 + \epsilon)^\ell \) and \( u_i^{rd} > 0 \). Let us consider the largest such \( \ell \) and denote by \( n_1 > y_\ell^{feas} \) the number of tasks satisfying both the aforementioned conditions.
Recall that in $H_{feas}$, $y_{\ell}^{feas}$ tasks $\tau_i$ with $v_{\ell i}^{rd} = \epsilon(1 + \epsilon)^\ell$ are assigned to type-2 processors. Therefore, it must be the case that in $H_{feas}$, some of the $m_1 - y_\ell^{feas}$ “additional” tasks were assigned to type-1 processors. Let $\tau_j$ denote one of these additional tasks, thus satisfying $v_{\ell j}^{rd} = \epsilon(1 + \epsilon)^\ell$ and $u_{\ell j}^{rd} = (1 + \epsilon)^x > 0$, for some $x \in [0, L - 1]$. Since this task $\tau_j$ has not been assigned yet by $A_{hvy}$, we know that at the time $A_{hvy}$ was assigning tasks in Step 1.3.1 with $\ell = x$, it went through case 1.3.1 and instead of choosing to assign $\tau_j$, it chose to assign another task $\tau_k \neq \tau_j$, also with $u_{\ell k}^{rd} = \epsilon(1 + \epsilon)^\ell$, that is assigned to type-2 processors in $H_{feas}$. Furthermore, according to case 1.3.1.3, it must hold that $v_{\ell j}^{rd} \geq v_{\ell k}^{rd} = \epsilon(1 + \epsilon)^\ell$. Now, two cases may arise.

1) If $v_{\ell j}^{rd} = v_{\ell k}^{rd} = \epsilon(1 + \epsilon)^\ell$ then $\tau_k$ is one of the $y_{\ell}^{feas}$ tasks assigned to type-2 processors in $H_{feas}$ and, since $A_{hvy}$ assigned $\tau_k$ to type-1 processors, there is a free “slot” on type-2 processors in which $\tau_j$ can fit. This contradicts our assumption that $\tau_j$ is unassigned at this time instant.

2) If $v_{\ell j}^{rd} > v_{\ell k}^{rd} = \epsilon(1 + \epsilon)^\ell$ then $\tau_k$ is one of the $y_{\ell}^{feas}$ tasks (with $r > \ell$) assigned to type-2 processors in $H_{feas}$ and, since $A_{hvy}$ assigned $\tau_k$ to type-1 processors, there was a free slot on type-2 processors in Step 1.3.2, when $\ell$ was equal to $r$. At this moment, when $\ell = r$, $A_{hvy}$ necessarily went through case 1.3.2.1 and since this case allows tasks with smaller utilization on type-2 processors to be accommodated in unused slots that were reserved for tasks with larger utilization, $\tau_j$ must have been assigned at that moment. This contradicts our assumption that $\tau_j$ is unassigned at this time instant.

Hence, we can conclude that $A_{hvy}$ does not declare failure for the ordered pair $P_{feas}$ of feasible configurations and the lemma holds for every task in $H_{int}^{feas}$.

Definition 5 (The corresponding sets $\Phi_{hvy}^1$ and $\Phi_{hvy}^2$). We define by $\Phi_{hvy}^1$ the set of tasks assigned to type-1 processors in $H_{feas}$ such that each task $\tau_k \in \Phi_{hvy}^1$ can be mapped to exactly one task $\tau_i \in \Gamma_{hvy}^1$ (bijection relation, implying $|\Phi_{hvy}^1| = |\Gamma_{hvy}^1|$) and for which $u_{\ell i} \leq (1 + \epsilon)u_{\ell k}$. The set $\Phi_{hvy}^2$ is defined analogously (for which $v_{\ell i} \leq (1 + \epsilon)v_{\ell k}$).

Lemma 2. After assigning the tasks in $H_{hvy}$, we have

$$\sum_{\tau_i \in \Gamma_{hvy}^1} u_{\ell i} \leq (1 + \epsilon)m_1$$

and

$$\sum_{\tau_i \in \Gamma_{hvy}^2} v_{\ell i} \leq (1 + \epsilon)m_2$$

Proof: We show only the proof of Expression (4), as the proof of Expression (5) is quite similar. For both cases, the proof is a direct consequence of Lemma 1. We know from Lemma 1 and Definition 5 that there exists a 1:1 mapping between every task $\tau_i \in \Gamma_{hvy}^1$ and every task $\tau_k \in \Phi_{hvy}^1$ such that $u_{\ell i} \leq (1 + \epsilon)u_{\ell k}$. Therefore, since $|\Phi_{hvy}^1| = |\Gamma_{hvy}^1|$ (from the bijective relation between the two sets), we have:

$$\sum_{\tau_i \in \Gamma_{hvy}^1} u_{\ell i} \leq (1 + \epsilon)\sum_{\tau_k \in \Phi_{hvy}^1} u_{\ell k}$$

Finally, we know from the feasibility of $H_{feas}$ that $\sum_{k \in \Phi_{hvy}^1} u_{\ell k} \leq m_1$ and hence $\sum_{\tau_i \in \Gamma_{hvy}^1} u_{\ell i} \leq (1 + \epsilon)m_1$.

V. ASSIGNING THE TASKS IN $H_{int}$ (STEP 2)

The tasks from $H_{hvy}$ that were not assigned by algorithm $A_{hvy}$ form the set $H_{int}$, i.e., $H_{int} = H_{hvy} \setminus \{\Gamma_{hvy}^1 \cup \Gamma_{hvy}^2\}$. Let us partition $H_{int}$ into two subsets $H_{int}^1$ and $H_{int}^2$ as follows:

$$H_{int}^1 = \{\tau_i \in H_{int} | u_{\ell i} < \epsilon \land v_{\ell i} \geq \epsilon\}$$

and

$$H_{int}^2 = \{\tau_i \in H_{int} | u_{\ell i} \geq \epsilon \land v_{\ell i} < \epsilon\}$$

A. The description of the algorithm $A_{int}$

The algorithm $A_{int}$ to assign the tasks in $H_{int}$ is as follows:

1) Assuming a platform, $\pi(1+\epsilon)$, assign all the tasks in $H_{int}^1$ to type-1 processors using the wrap-around technique. This technique works as follows. Take the first processor of type-1 and assign as many of the tasks as possible from $H_{int}^1$ “integrally” onto that processor. When a task fails to be assigned integrally, assign that task “fractionally” such that the current processor is filled completely and the remaining fraction is assigned to the next processor of type-1, continue this procedure until all the tasks from $H_{int}^1$ are assigned to type-1 processors.

2) Analogously, assign all the tasks in $H_{int}^2$ to type-2 processors using the wrap-around technique.

B. Assignment analysis

We now show that for a task set $\pi$ that is feasible on a platform $\pi(1+\epsilon)$, $A_{int}$ always succeeds in assigning all the tasks in $H_{int}^1$ to type-1 processors on a platform $\pi(1+\epsilon)$. That is, if $\Gamma_{int}^1$ and $\Gamma_{int}^2$ denote the set of tasks assigned to type-1 and type-2 processors by $A_{int}$, we have $\Gamma_{int}^1 = \Gamma_{hvy}^1$ and $\Gamma_{int}^2 = \Gamma_{hvy}^2$.

In the following lemma, we make use of the fact that the two sets of tasks $\Gamma_{hvy}^1$ and $\Gamma_{hvy}^2$ have been obtained by algorithm $A_{hvy}$, using the ordered pair $P_{feas}$ of feasible configurations.

Lemma 3. After assigning all the tasks in $H_{int}$ using the ordered pair of feasible configuration, we have:

$$\sum_{\tau_i \in \Gamma_{int}^1} u_{\ell i} + \sum_{\tau_i \in \Gamma_{int}^2} u_{\ell i} \leq (1 + \epsilon)m_1$$

and

$$\sum_{\tau_i \in \Gamma_{int}^1} v_{\ell i} + \sum_{\tau_i \in \Gamma_{int}^2} v_{\ell i} \leq (1 + \epsilon)m_2$$

Proof: In the feasible assignment $H_{feas}$, $|\Gamma_{int}^1|$ number of tasks with $u_{\ell i} < \epsilon$ and $v_{\ell i} \geq \epsilon$ must have been assigned to type-1 processors. This is a consequence of the fact that $P_{feas}$ contains exactly the same number of tasks with utilization $\geq \epsilon$ on the processor that they are assigned to, as in $H_{feas}$. Let $\Gamma_{int}$ denote the set of tasks with $u_{\ell i} < \epsilon$ and $v_{\ell i} \geq \epsilon$ that are assigned to type-1 processors in $H_{feas}$. Since $H_{feas}$ is a feasible assignment, it holds that,

$$\sum_{\tau_i \in \Gamma_{int}^1} u_{\ell i} + \sum_{\tau_i \in \Gamma_{int}} u_{\ell i} \leq m_1$$

Since the number of tasks with $u_{\ell i} < \epsilon$ and $v_{\ell i} \geq \epsilon$ that have been assigned to type-1 processors is same in both $H_{feas}$ and the assignment computed by our algorithm, we have $|\Gamma_{int}^1| = |\Gamma_{int}| = |\Gamma_{int}^1|$. Here, it is worth recalling Step 1.3.1.3 and Step 1.3.2.4 of algorithm $A_{hvy}$. In these steps, while assigning the tasks to processors of type-1 (resp., type-2), when $A_{hvy}$ has to choose few tasks to assign from the available set of tasks, it always chooses those tasks that
have a larger utilization on type-2 (resp., type-1) processors (leaving "easier" tasks for $A_{\text{int}}$ to assign). Now coming back to algorithm $A_{\text{int}}$, although the tasks (with $u_i < \epsilon$ and $v_i \geq \epsilon$) assigned by $A_{\text{int}}$ to type-1 processors may not be the same as those assigned by $H_{\text{ideal}}$, we can infer (using the rules of Step 1.3.1.3 and Step 1.3.2.1 of $A_{\text{hvy}}$) that:

$$\sum_{\tau \in \Gamma_{\text{hvy}}} u_i \leq \sum_{\tau \in \phi_{\text{hvy}}} u_i$$  

Applying Inequality (6) and (12) on (11) and then performing some arithmetic manipulations (see [15] for details), we get:

$$\sum_{i \in \Gamma_{\text{hvy}}} u_i + \sum_{i \in \Gamma_{\text{int}}} u_i \leq (1 + \epsilon) \times m_1$$

Using similar reasoning as above, we can show that Expression (10) holds as well. Hence the proof.

Corollary 1. After assigning the tasks in $\tau_{\text{int}}$, we have:

$$\sum_{\tau \in \Gamma_{\text{hvy}} \cup \Gamma_{\text{int}}} u_i \leq (1 + \epsilon) \sum_{\tau \in \phi_{\text{hvy}} \cup \phi_{\text{int}}} u_i$$  

and

$$\sum_{\tau \in \Gamma_{\text{hvy}} \cup \Gamma_{\text{int}}} v_i \leq (1 + \epsilon) \sum_{\tau \in \phi_{\text{hvy}} \cup \phi_{\text{int}}} v_i$$

Proof: Inequality (13) follows from Expressions (6) and (12) (since $\Gamma_{\text{int}} = \Gamma_{\text{int}}^1$) and Inequality (14) can be inferred from analogous expressions for type-2 processors.

VI. ASSIGNING THE TASKS IN $\tau_{\text{hvy}}$ (STEP 3)

Let us partition $\tau_{\text{int}}$ into $\tau_{\text{int}}^1$ and $\tau_{\text{int}}^2$ as follows:

$$\tau_{\text{int}}^1 = \{ \tau_1 \in \tau_{\text{int}} | u_i \leq v_i \}$$  

$$\tau_{\text{int}}^2 = \{ \tau_i \in \tau_{\text{int}} | u_i > v_i \}$$

A. The description of the algorithm $A_{\text{hvy}}$

The pseudo-code for assigning tasks in $\tau_{\text{int}}$ is shown in Algorithm 1 (which uses the $\text{fract-next-fit}$ subroutine shown in Algorithm 2). The intuition behind the design of this algorithm is that, assuming a platform, $\tau_{\text{hvy}}^{1,2}$, first we assign tasks to processors on which they have a smaller utilization (lines 1 and 2). Then, if there are remaining tasks, these have to be assigned to processors on which they have a larger utilizations (lines 7 and 15).

B. Assignment analysis

First, we present some useful result in Lemma 4 obtained by relating the problem under consideration to the fractional knapsack problem (see Chapter 16.2 in [16]). This result will be used in Lemma 5. The relation between the fractional knapsack problem and the problem under consideration was explored in [13]. Lemma 4 is an adaptation of Lemma 5 in [13]. Hence, we only state the lemma here. The detailed description of the fractional knapsack problem, its relation with

Algorithm 1: $A_{\text{hvy}}$: An algorithm to assign $\tau_{\text{hvy}}$ tasks

1. $\Gamma_{\text{hvy}}^1 := \text{fract-next-fit}(\Gamma_{\text{hvy}}^1, m_1)$
2. $\Gamma_{\text{hvy}}^2 := \text{fract-next-fit}(\Gamma_{\text{hvy}}^2, m_2)$
3. if $(\Gamma_{\text{hvy}}^1 = \tau_{\text{hvy}}^1 \land \Gamma_{\text{hvy}}^2 = \tau_{\text{hvy}}^2)$ then declare SUCCESS
4. if $(\Gamma_{\text{hvy}}^1 \neq \tau_{\text{hvy}}^1 \land \Gamma_{\text{hvy}}^2 \neq \tau_{\text{hvy}}^2)$ then declare FAILURE
5. if $(\Gamma_{\text{hvy}}^1 \neq \tau_{\text{hvy}}^1 \land \Gamma_{\text{hvy}}^2 = \tau_{\text{hvy}}^2)$ then

   $\Gamma_{\text{hvy}} := \Gamma_{\text{hvy}}^2 \backslash \tau_{\text{hvy}}^2$
6. if $(\Gamma_{\text{hvy}}^1 = \tau_{\text{hvy}}^1 \land \Gamma_{\text{hvy}}^2 \neq \tau_{\text{hvy}}^2)$ then

   $\Gamma_{\text{hvy}} := \Gamma_{\text{hvy}}^1 \backslash \tau_{\text{hvy}}^1$
7. if $(\text{fract-next-fit}(\Gamma_{\text{hvy}}^2, m_2) = \Gamma_{\text{hvy}}^2)$ then
   | declare SUCCESS
8. else
   | declare FAILURE
9. end
10. if $(\text{fract-next-fit}(\Gamma_{\text{hvy}}^1, m_1) = \Gamma_{\text{hvy}}^1)$ then
   | declare SUCCESS
11. else
   | declare FAILURE
12. end
13. end

Algorithm 2: fract-next-fit($\Gamma$, $ps$): Next-fit bin-packing with fractional assignment of tasks

Input : $ts$: set of tasks; $ps$: set of processors
Output: set of tasks that were assigned successfully

1. If $ps$ consists of type-1 (resp., type-2) processors, then sort $ts$ by decreasing $v_i/u_i$ (resp., increasing $v_i/u_i$). Use any order for processors $ps$, but maintain it during the execution of $\text{fract-next-fit}$.
2. Assign tasks using $\text{wrap-around}$ technique\(^2\).
3. Return the set of successfully assigned tasks.

the problem under consideration and the proof of Lemma 4 can be found in Appendix A in [15].

Lemma 4. Consider a task set $T$ and a two-type platform conforming to the system model of Section II. Let us partition $T$ into two disjoint subsets, $T^1$ and $T^2$ as follows:

$$T = T^1 \cup T^2$$

$$\forall \tau_i \in T^1 : u_i \leq v_i$$

$$\forall \tau_i \in T^2 : u_i > v_i$$

Let $x$ denote a real number such that $0 \leq x \leq m_1$.

Let $A_1$ denote a subset of $T^1$ such that $\sum_{\tau_{\text{int}} \in A_1} u_i > m_1 - x$ and for every pair of tasks $\tau_i \in A_1$ and $\tau_j \in T^1 \backslash A_1$ it holds that $\frac{u_i}{u_j} - 1 \geq \frac{m_1}{m_1 - x - 1}$.

Let $A_2$ denote $T^1 \backslash A_1$.
Let $B_1$ denote a subset of $T^1$ such that $\sum_{\tau_{\text{int}} \in B_1} u_i \leq m_1 - x$.
Let $B_2$ denote $T^1 \backslash B_1$. It then holds that:

$$\sum_{\tau_i \in A_1} u_i + \sum_{\tau_i \in A_2} u_i + \sum_{\tau_i \in B_1} v_i + \sum_{\tau_i \in B_2} v_i \leq \sum_{\tau_i \in A_1} u_i + \sum_{\tau_i \in B_2} v_i \leq (1 + 2\epsilon)m_1$$

Lemma 5. Let $\Gamma_{\text{hvy}}$ and $\Gamma_{\text{hvy}}^2$ be the subset of tasks from $\tau_{\text{hvy}}$ that are assigned by $A_{\text{hvy}}$ to type-1 and type-2 processors, respectively. After assigning all the tasks from $\tau_{\text{hvy}}$, we have:

$$\sum_{\tau_i \in \Gamma_{\text{hvy}}} u_i + \sum_{\tau_i \in \Gamma_{\text{hvy}}^1} u_i + \sum_{\tau_i \in \Gamma_{\text{hvy}}^1} u_i \leq (1 + 2\epsilon)m_1$$

and

$$\sum_{\tau_i \in \Gamma_{\text{hvy}}} v_i + \sum_{\tau_i \in \Gamma_{\text{hvy}}^2} v_i + \sum_{\tau_i \in \Gamma_{\text{hvy}}^2} v_i \leq (1 + 2\epsilon)m_2$$
where

1) all the tasks in \( \tau_{\text{hvy}} \setminus \tau_{\text{int}} \) are assigned integrally
2) some tasks in \( \tau_{\text{int}} \) are assigned fractionally and the rest are assigned integrally
3) some tasks in \( \tau_{\text{lgf}} \) are assigned fractionally and the rest are assigned integrally

**Proof:** Informally, the claim can be written as follows: if there exists a feasible partitioning for a task set \( \tau \) on a two-type platform \( \pi \) then algorithms \( A_{\text{hvy}} \), \( A_{\text{int}} \) and \( A_{\text{lgf}} \) succeed in assigning the tasks in \( \tau \) on a platform \( \pi^{(1+\epsilon)} \), with some tasks assigned fractionally. We already know from Lemma 3 that after assigning the tasks in \( \tau_{\text{hvy}} \setminus \tau_{\text{int}} \) and \( \tau_{\text{int}} \) using algorithms \( A_{\text{hvy}} \) and \( A_{\text{int}} \), respectively, the sum of the utilizations of the tasks assigned on type-1 (resp., type-2) processors does not exceed \((1 + \epsilon)m_1\) (resp., \((1 + \epsilon)m_2\)).

Therefore, we need to show that after assigning the tasks in \( \tau_{\text{int}} \), by using algorithm \( A_{\text{lgf}} \), the sum of the utilizations of the tasks assigned on processors of type-1 (resp., type-2) does not exceed \((1 + 2\epsilon)m_1\) (resp., \((1 + 2\epsilon)m_2\)). An equivalent claim is that, after assigning tasks in \( \tau_{\text{hvy}} \setminus \tau_{\text{int}} \) and \( \tau_{\text{int}} \) by using algorithms \( A_{\text{hvy}} \) and \( A_{\text{int}} \) respectively, if \( A_{\text{lgf}} \) fails to assign the tasks in \( \tau_{\text{lgf}} \) (with fractional assignment of tasks allowed) on platform \( \pi^{(1+\epsilon)} \), then there does not exist a feasible partitioning of the tasks in \( \tau \) on platform \( \pi \). Here, we prove this equivalent claim by contradiction. Assume that there exists a feasible assignment \( H_{\text{feas}} \) of \( \tau \) on \( \pi \) but \( A_{\text{lgf}} \) fails to assign the tasks in \( \tau_{\text{lgf}} \) on \( \pi^{(1+\epsilon)} \) (after \( A_{\text{hvy}} \) and \( A_{\text{int}} \) successfully assigned the tasks of \( \tau_{\text{hvy}} \setminus \tau_{\text{int}} \) and \( \tau_{\text{int}} \)).

Since \( A_{\text{lgf}} \) failed to assign these tasks, it must have declared failure and we explore all possibilities for this to occur:

**Failure on line 4 in Algorithm 1:** From the case, we have \( \Gamma_{\text{lgf}}^1 \subset \Gamma_{\text{lgf}}^1 \) and \( \Gamma_{\text{lgf}}^2 \subset \Gamma_{\text{lgf}}^1 \).

We know that the tasks assigned to type-2 processors at this stage are \( \Gamma_{\text{hvy}}^2 \cup \Gamma_{\text{int}}^2 \cup \Gamma_{\text{lgf}}^2 \) and a strict subset of tasks from \( \Gamma_{\text{lgf}}^1 \) (line 7). Therefore, we can rewrite Expression (30) as:

\[
\sum_{r \in \Gamma_{\text{hvy}}^2 \cup \Gamma_{\text{int}}^2 \cup \Gamma_{\text{lgf}}^2} v_i > m_{\text{max}}(1 + 2\epsilon) - v_f
\]

Since \( v_f \in \Gamma_{\text{lgf}}^1 \setminus \Gamma_{\text{lgf}}^1 \), we know that \( v_f < \epsilon \leq m_{\text{max}} \). Using this on Expression (31), then applying Expression (14) and finally performing some arithmetic manipulations (see [15] for details) gives us:

\[
\sum_{r \in \Gamma_{\text{hvy}}^2 \cup \Gamma_{\text{int}}^2 \cup \Gamma_{\text{lgf}}^2} v_i > m_2(1 + 2\epsilon) - v_f
\]

We also know that, when \( A_{\text{lgf}} \) executed line 1 (where it performed \( \text{fract-next-fit} \)), there must have been a task \( v_f \in \Gamma_{\text{lgf}}^1 \setminus \Gamma_{\text{lgf}}^1 \) which was attempted on type-1 processors but failed to be assigned. Note that this task \( v_f \) may be the same as \( v_f \) mentioned above or it may be different. Because it was not possible to assign \( v_f \) on type-1 processors, we know that:

\[
\sum_{p \in \Gamma_{\text{hvy}}^1 \cup \Gamma_{\text{int}}^1} U[p] + u_{f_1} > m_1(1 + 2\epsilon)
\]

We know that the tasks assigned to type-1 processors are \( \Gamma_{\text{hvy}}^1 \cup \Gamma_{\text{int}}^1 \cup \Gamma_{\text{lgf}}^1 \) and thus, we rewrite Expression (33) as:

\[
\sum_{r \in \Gamma_{\text{hvy}}^1 \cup \Gamma_{\text{int}}^1 \cup \Gamma_{\text{lgf}}^1} u_i > m_1(1 + 2\epsilon) - u_{f_1}
\]

Since \( v_f \in \Gamma_{\text{lgf}}^1 \setminus \Gamma_{\text{lgf}}^1 \), we have \( u_{f_1} < \epsilon \leq 2\epsilon \). Using this on Expression (34), then applying Expression (13) and finally performing some arithmetic manipulations (see [15] for details) gives us:
Finally, Expression (35) can be rewritten as:

\[
\sum_{\tau_i \in \Phi_1^{\text{int}}} u_i + \sum_{\tau_i \in \Phi_2^{\text{int}}} u_i > m_1
\]

Expression (37) can be rewritten as:

\[
\sum_{\tau_i \in \Phi_1^{\text{int}}} u_i \leq m_1 - \left( \sum_{\tau_i \in \Phi_1^{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi_1^{\text{int}}} u_i \right)
\]

We can now reason about the inequalities we obtained about the assignment \(H_{\text{feas}}\) and the one constructed by \(A_{\text{frac}}\). We can see that Expressions (36) and (39), with \(x = \sum_{\tau_i \in \Phi_1^{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi_1^{\text{int}}} u_i\), ensure that the assumptions of Lemma 4 are true, given the ordering of tasks in \(\tau_{\text{int}}\) during assignment over type-1 processors (line 1 in Algorithm 2), which ensures that \(\forall \tau_i \in \Gamma_1^{\text{hvy}}, \forall \tau_j \in \Gamma_2^{\text{hvy}}: \frac{u_i}{u_j} \geq \frac{v_i}{v_j}\). By applying Lemma 4 with the following input:

- \(T = \tau \setminus (\Phi_1^{\text{hvy}} \cup \Phi_2^{\text{hvy}} \cup \Phi_1^{\text{int}} \cup \Phi_2^{\text{int}})\),
- \(T^1 = \tau_{\text{frac}}^{\text{int}}, T^2 = \tau_{\text{frac}}^{\text{hvy}},\)
- \(x = \sum_{\tau_i \in \Phi_1^{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi_1^{\text{int}}} u_i,\)
- \(A_1 = \Gamma_1^{\text{hvy}}, A_2 = \Gamma_2^{\text{hvy}}\),
- \(B_1 = \Phi_1^{\text{int}}, B_2 = \Phi_2^{\text{int}}\),
- \(A_1 = \Gamma_1^{\text{hvy}}, A_2 = \Gamma_2^{\text{hvy}}\),
- \(B_1 = \Phi_1^{\text{int}}, B_2 = \Phi_2^{\text{int}}\),

we get:

\[
\sum_{\tau_i \in \Phi_1^{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi_1^{\text{int}}} u_i + \sum_{\tau_i \in \Phi_2^{\text{int}}} v_i \leq \sum_{\tau_i \in \Phi_1^{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi_1^{\text{int}}} u_i + \sum_{\tau_i \in \Phi_2^{\text{int}}} v_i
\]

Adding \(\sum_{\tau_i \in \Phi_1^{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi_2^{\text{int}}} v_i \) to both sides of the above inequality, then applying Expressions (37) and (38) to the right-hand side and then applying Expressions (32) and (35) to the left-hand side yields:

\[m_1 + m_2 < m_1 + m_2\]

This is a contradiction.

**Failure on line 18 in Algorithm 1**: A contradiction results — proof analogous to the previous case.

We showed that all the cases where \(A_{\text{frac}}\) declares FAILURE lead to a contradiction. Hence, the lemma holds.

**VII. INTEGRAL ASSIGNMENT OF \(\tau_{\text{frac}}\) AND \(\tau_{\text{hvy}}\) (STEP 4)**

We now discuss how to integrally assign the tasks from \(\tau_{\text{frac}}\) and \(\tau_{\text{hvy}}\) that were fractionally assigned by algorithms \(A_{\text{frac}}\) and \(A_{\text{hvy}}\), respectively. We also show that, if there is a feasible partitioning of the given task set on a given two-type platform then our PTAS succeeds in finding such a feasible partitioning on a platform in which each processor is \(1 + 3\epsilon\) times faster.

**A. The description of the algorithm \(A_{\text{frac}}\)**

The algorithm, \(A_{\text{frac}}\), works as follows:

1) copy the assignment (made by \(A_{\text{hvy}}, A_{\text{frac}}\) and \(A_{\text{int}}\)) onto a faster platform \(\pi^{(1+3\epsilon)}\).

2) on this platform, \(\pi^{(1+3\epsilon)}\), assign a task split between any two processors \(p_1\) and \(p_1 + 1\) of type-1 entirely on to processor \(p_1\), where \(1 \leq p_1 < m_1\); similarly, assign a task split between any two processors \(p_2\) and \(p_2 + 1\) of type-2 entirely on to processor \(p_2\), where \(1 \leq p_2 < m_2\).

**B. Assignment analysis**

**Theorem 1.** If there exists a feasible partitioning of \(\tau\) on \(\pi\) then our PTAS algorithm, PTASNF, (which uses \(A_{\text{hvy}}, A_{\text{int}}, A_{\text{frac}}\) and \(A_{\text{frac}}\) in sequence) succeeds in finding a feasible partitioning of \(\tau\) on \(\pi^{(1+3\epsilon)}\).

**Proof:** We know from Lemma 5 that if there exists a feasible partitioning of \(\tau\) on \(\pi\) then the three algorithms \(A_{\text{hvy}}, A_{\text{frac}}\) and \(A_{\text{int}}\) described in Sections IV-VI succeed in assigning tasks in \(\tau\) (with a subset of tasks from \(\tau_{\text{int}}\) and \(\tau_{\text{frac}}\) fractionally assigned) on \(\pi^{(1+\epsilon)}\). As a consequence, we have:

\[\forall p \in \pi^{(1+\epsilon)}: U[p] \leq 1 + 2\epsilon\]  

(40)

We also know that in such an assignment (as a consequence of using the wrap-around technique in \(A_{\text{int}}\) and \(A_{\text{frac}}\)):

- at most \(m_1 - 1\) tasks are split between processors of type-1 with one task split between each pair of consecutive processors; let the set \(S_1^{\text{split}}\) denote these fractional tasks.
- at most \(m_2 - 1\) tasks are split between processors of type-2 with one task split between each pair of consecutive processors; let the set \(S_2^{\text{split}}\) denote these fractional tasks.
- at most one task (from \(\tau_{\text{frac}}\)) is split between processors of type-1 and type-2; let \(T^{\text{split}} \in \tau_{\text{frac}}\) denote this task that must be split between the \(m_1^{\text{th}}\) processor of type-1 and the \(m_2^{\text{th}}\) processor of type-2.
- the rest of the tasks are integrally assigned to either type-1 or type-2 processors.

Let \(T_{p_1, p_1+1} \in S_1^{\text{split}}\) denote the task split between the \(p_1^{\text{th}}\) and the \((p_1 + 1)^{\text{th}}\) processors of type-1 where \(1 \leq p_1 < m_1\). Analogously, let \(T_{p_2, p_2+1} \in S_2^{\text{split}}\) denote the task split between the \(p_2^{\text{th}}\) and the \((p_2 + 1)^{\text{th}}\) processors of type-2 where \(1 \leq p_2 < m_2\).

To prove the theorem, we need to show that \(A_{\text{frac}}\) succeeds in integrally assigning all the fractional tasks on \(\pi^{(1+3\epsilon)}\).

On Step 1, \(A_{\text{frac}}\) copies the feasible assignment (retaining the fractional task assignments) onto the faster platform \(\pi^{(1+3\epsilon)}\). After this step,

\[\forall p \in \pi^{(1+3\epsilon)}: U[p] \leq 1 + 2\epsilon\]  

(41)

Since \(\Gamma_1^{\text{split}} \subseteq \{\tau_1^{\text{int}} \cup \tau_{\text{frac}}\}\), \(\Gamma_2^{\text{split}} \subseteq \{\tau_2^{\text{int}} \cup \tau_{\text{frac}}\}\), we have:
(2), (7) \Rightarrow \forall \tau_i \in \Gamma_1^1 : u_i < \epsilon \tag{42}
(2), (8) \Rightarrow \forall \tau_i \in \Gamma_2^1 : v_i < \epsilon \tag{43}

On Step 2, \(A_{\text{frac}}\) assigns the split tasks integrally. So, \(\forall p_1 \in \text{type-1 of } \pi^{(1+3\epsilon)}\), it moves the fraction of the task \(\tau_{p_1, p_1+1}\) that is assigned to \((p_1 + 1)\text{th processor of type-1 to } p_1\text{th processor of type-1}. After this re-assignment, it follows from Expressions (41) and (42) that:

\[
\forall p_1 \in \text{type-1 of } \pi^{(1+3\epsilon)} \wedge p_1 \neq m_1 : \quad U[p_1] \leq 1 + 3\epsilon \tag{44}
\]

if \(p_1 \in \text{type-1 of } \pi^{(1+3\epsilon)} \wedge p_1 = m_1 : \quad U[p_1] \leq 1 + 2\epsilon \tag{45}

Analogously, \(\forall p_2 \in \text{type-2 of } \pi^{(1+3\epsilon)}\), it moves the fraction of the task \(\tau_{p_2, p_2+1}\) that is assigned to \((p_2 + 1)\text{th processor of type-2 to } p_2\text{th processor of type-2}. After this re-assignment, it follows from Expressions (41) and (43) that:

\[
\forall p_2 \in \text{type-2 of } \pi^{(1+3\epsilon)} \wedge p_2 \neq m_2 : \quad U[p_2] \leq 1 + 3\epsilon \tag{46}
\]

if \(p_2 \in \text{type-2 of } \pi^{(1+3\epsilon)} \wedge p_2 = m_2 : \quad U[p_2] \leq 1 + 2\epsilon \tag{47}

Finally, the task \(\tau_f\) that is split between the \(m_1\text{th processor of type-1 and the } m_2\text{th processor of type-2 remains to be integrally assigned. Since } \tau_f \in \tau_{\text{split}}\), it holds that \(u_f < \epsilon\) and \(v_f < \epsilon\). From Expression (45) and (47), it follows that task \(\tau_f\) can be integrally assigned to either \(m_1\text{th or } m_2\text{th processor}. Hence, after integrally assigning this task, we obtain:

\[
\forall p \in \pi^{(1+3\epsilon)} : U[p] \leq 1 + 3\epsilon \tag{48}
\]

Since Expression (48) is a necessary and sufficient schedulability condition for EDF on a uniprocessor of capacity \(1 + 3\epsilon\), the assignment of \(T\) on \(\pi^{(1+3\epsilon)}\) returned by our algorithm, \(\text{PTAS}_{\text{NF}}\), is a feasible assignment. Hence, the proof.

VIII. EXPERIMENTAL SETUP AND RESULTS

After studying the theoretical (worst-case) bound, i.e., the approximation ratio of our algorithm, \(\text{PTAS}_{\text{NF}}\), we evaluate its average-case performance and compare it with prior state-of-the-art, i.e., \(\text{PTAS}_{\text{LP}}\). For this purpose, we look at the following aspects: (i) how much faster processors our algorithm needs in practice in order to successfully partition a task set compared to \(\text{PTAS}_{\text{LP}}\)? and (ii) how fast our algorithm runs compared to \(\text{PTAS}_{\text{LP}}\)? Also, we look at (iii) how much pessimism is there in our theoretically derived performance bound? In order to answer these questions, we performed two sets of experiments. The first set of experiments described in Section VIII-A addresses (i) and (ii) and the second set of experiments described in Section VIII-B addresses (iii).

A. Comparison with prior state-of-the-art

We compare the average-case performance of \(\text{PTAS}_{\text{NF}}\) with \(\text{PTAS}_{\text{LP}}\). We implemented both the algorithms in C on an Intel Core2 (2.80 GHz) machine. For \(\text{PTAS}_{\text{LP}}\), we used a state-of-the-art LP solver, IBM ILOG CPLEX [17].

For a given task set, we define the minimum required speedup factor, \(\text{MRSF}_{\text{NF}}\), of \(\text{PTAS}_{\text{NF}}\) as the minimum amount of extra speed of processors that \(\text{PTAS}_{\text{NF}}\) needs, so as to succeed in finding a feasible partitioning as compared to an optimal algorithm. We define \(\text{MRSF}_{\text{LP}}\) of \(\text{PTAS}_{\text{LP}}\) analogously. For different values of \(\epsilon\), we assess the average-case performance of these algorithms by measuring their (i) minimum required speedup factors and (ii) running times.

The problem instances (number of tasks, their utilizations and number of processors of each type) were generated randomly. Each problem instance had at most 10 tasks and at most 2 processors of each type. For performing fair evaluation, we convert the randomly generated task sets into critically feasible task sets — more details in Appendix B in [15]. A task set is termed critically feasible if it is feasible on a given two-type platform but rendered infeasible if all \(u_i\) and \(v_i\) are increased by an arbitrarily small factor. The intuition behind using critically feasible task sets in our simulations is that it is “hard” to find a feasible partitioning for these task sets since only a few assignments are feasible among all the possible assignments. Hence, by using such task sets, we believe our evaluations have been fair and unbiased.

We ran \(\text{PTAS}_{\text{NF}}\) and \(\text{PTAS}_{\text{LP}}\) on 5000 critically feasible task sets and for each task set, we obtain \(\text{MRSF}_{\text{NF}}\) and \(\text{MRSF}_{\text{LP}}\) as follows. We initially set the speedup factor to 1.0 and input the task set to the algorithm. If the algorithm cannot find a feasible mapping, we increment the speedup factor by a small value, i.e., by 0.01, and divide the original utilizations, \(u_i\) and \(v_i\), of each task by the new speedup factor (to simulate the faster platform) and feed the resulting task set to the algorithm. These steps (adjust the speedup factor and feed back the derived task set) are repeated till the algorithm succeeds, which gives us the MRSF of the algorithm for the task set. This entire procedure is repeated for 5000 task sets.

With this procedure, we obtain the histograms of MRSFs for both the algorithms for different values of \(\epsilon\). Figure 1 shows the histogram for \(\epsilon = 0.2\) (note that the y-axis is in log scale). As can be seen, the MRSF of \(\text{PTAS}_{\text{NF}}\) never exceeded 1.12 which is 20% from the optimal value of 1.0 compared to its upper bound of \(1 + 3\epsilon = 1.60\), i.e., \(\frac{1.12 - 1.0}{1.60 - 1.0} \times 100 = 20\%\), whereas \(\text{MRSF}_{\text{LP}}\) is as high as 1.30 which is 60% from the optimal value of 1.0 compared to its upper bound of \(1 + \frac{1}{\epsilon} = 1.50\), i.e., \(\frac{1.30 - 1.0}{1.50 - 1.0} \times 100 = 60\%\). Therefore, \(\text{PTAS}_{\text{NF}}\) requires much smaller processor speedup on an average than \(\text{PTAS}_{\text{LP}}\) in order to find a feasible partitioning. The observations for other values of \(\epsilon\) follow the same trend — see Appendix B in [15].

We also measure the average running times of both algorithms for different values of \(\epsilon\). In these experiments, the speedup factor is set to \(1 + 3\epsilon\) for \(\text{PTAS}_{\text{NF}}\) and to \(1 + \frac{1}{\epsilon}\) for \(\text{PTAS}_{\text{LP}}\). This ensures that both the algorithms succeed in finding a feasible partitioning for a given task set in a single run and hence the experiments are not biased to give advantage to any of them. In our experiments with 5000 task sets, as can be seen in Table I, for \(\epsilon = 0.1\), \(\text{PTAS}_{\text{NF}}\) runs approximately 50000 times faster compared to \(\text{PTAS}_{\text{LP}}\). This factor is even higher for other values of \(\epsilon\).

To summarize, our algorithm exhibits a better average-case performance by requiring significantly smaller processor speedup for finding a feasible partitioning and by running orders of magnitude faster compared to \(\text{PTAS}_{\text{LP}}\). Overall, \(\text{PTAS}_{\text{NF}}\) outperforms prior state-of-the-art, \(\text{PTAS}_{\text{LP}}\).
and size of each problem compared to previous set of experiments.

type platform in which each processor is
on a two-type platform then given an
— see Appendix B in [15].

The observations for other values of
the task sets, the factor did not exceed
theoretical bound (i.e.,
MRSF
PTAS
sets where each task set had at most
number of problem instances with each problem instance being
different values of
the analysis of
TABLE I: Comparison of run times of PTAS\textsubscript{NF} and PTAS\textsubscript{LP} (\mu s).

<table>
<thead>
<tr>
<th>Value of $\epsilon$</th>
<th>Measured avg. run-time</th>
<th>Ratio of avg. run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10</td>
<td>1.28.57</td>
<td>6583384.71</td>
</tr>
<tr>
<td>0.15</td>
<td>45.43</td>
<td>6914227.72</td>
</tr>
<tr>
<td>0.20</td>
<td>18.40</td>
<td>4490601.29</td>
</tr>
<tr>
<td>0.25</td>
<td>10.38</td>
<td>1564060.39</td>
</tr>
<tr>
<td>0.30</td>
<td>7.17</td>
<td>465894.09</td>
</tr>
</tbody>
</table>

B. Evaluation of PTAS\textsubscript{NF} for different values of $\epsilon$

In order to understand how much pessimism is there in the analysis of PTAS\textsubscript{NF}, we evaluated its performance for different values of $\epsilon$. In this set of experiments, we chose larger number of problem instances with each problem instance being more complex\textsuperscript{3}. We generated 10000 critically feasible task sets where each task set had at most 25 tasks and at most 3 processors of each type. Then, for different values of $\epsilon$, we ran PTAS\textsubscript{NF} on these 10000 critically feasible task sets and obtain the histogram of MRSF\textsubscript{NF}. Figure 2 shows the histogram for $\epsilon = 0.3$. As can be seen, for almost 98% of the task sets, the MRSF\textsubscript{NF} did not exceed 1.06, i.e., approximately 7% of its theoretical bound (i.e., $1+3\epsilon = 1.90$), for the remaining 2% of the task sets, the factor did not exceed 1.12, i.e., approximately 13% of its theoretical bound. Thus, in the simulations, for the vast majority of task sets, our algorithm requires much smaller processor speedup than indicated by its approximation ratio. The observations for other values of $\epsilon$ follow the same trend — see Appendix B in [15].

Hence, PTAS\textsubscript{NF} performs significantly better in simulations than indicated by its theoretical bound.

IX. Conclusions

A polynomial-time approximation scheme was proposed for the problem of partitioning a given collection of implicit-deadline sporadic tasks upon a multiprocessor platform in which there are two distinct kinds of processors. It provides the following guarantee: if a task set has a feasible partitioning on a two-type platform then given an $\epsilon$, our PTAS succeeds in finding such a feasible partitioning for the task set on a two-type platform in which each processor is $1+3\epsilon$ times faster.

\textsuperscript{3}Since we do not run PTAS\textsubscript{LP} (which takes much longer to output the solution) in this batch of experiments, we could increase the problem instances and size of each problem compared to previous set of experiments.

In simulations, our algorithm outperforms prior state-of-the-art PTAS [1] and also performs significantly better in simulations than indicated by its theoretical bound.

ACKNOWLEDGMENT

This work was partially supported by FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme “Thematic Factors of Competitiveness”), within REHEAT project, ref. FCOMP-01-0124-FEDER-00045; by FCT and the EU ARTEMIS JU funding, within RECOMP project, ref. ARTEMIS/0202/2009, JU grant nr. 100202; by FCT and ESP through POPH, under PhD grant SFRH/BD/66771/2009.

REFERENCES

In this section, we give a detailed description of the fractional knapsack problem followed by its relation with the task assignment problem on two-type platform and finally provide the proof of Lemma 4 which we used earlier (in Lemma 5) while proving the approximation ratio of our algorithm, PTAS_{NF}.

1) Fractional knapsack problem: We now define the fractional knapsack problem.

**Definition 6 (Fractional knapsack problem).** A vector \( x \) has \( n \) elements. The problem instance is represented by vectors \( p \) and \( w \) of real numbers, arranged such that \( \frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \) \( \forall i \in \{1,2,...,n-1\} \). (Intuitively, \( p_i \) and \( w_i \) may be thought of as, respectively, the “profit” and “weight” of an item, indexed \( i \), while \( x_i \) as the fraction of it that is employed.) Consider the problem of assigning profits to the elements in \( x \) so as to:

maximize \( \sum_{i=1}^{n} x_i \cdot p_i \) subject to

\[ \sum_{i=1}^{n} x_i \cdot w_i \leq \text{CAP} \]

and \( x_i \) is a real number and \( 0 \leq x_i \leq 1 \)

and \( \text{CAP} \) is a given upper bound.

Intuitively, determine how much of each item to use such that cumulative profit is maximized, subject to cumulative weight not exceeding some bound.

**Lemma 6.** The following algorithm optimally solves the fractional knapsack problem:

**Algorithm 3:** An optimal algorithm for the fractional knapsack problem

1. re-index tuples \( \{p_i, w_i\} \) by order of descending \( p_i/w_i \)
2. for \( i:=1 \) to \( n \) do
3. \[ x_i:=0 \]
4. end
5. \( i:=1 \), \( \text{SUMWEIGHT}=0 \), \( \text{SUMPROFIT}=0 \)
6. while \( (\text{SUMWEIGHT}+w_i \leq \text{CAP}) \) and \( (i \leq n) \) do
7. \[ x_i:=1 \]
8. \[ \text{SUMWEIGHT}:=\text{SUMWEIGHT}+w_i \]
9. \[ \text{SUMPROFIT}:=\text{SUMPROFIT}+p_i \]
10. \( i:=i+1 \)
11. end
12. if \( i \leq n \) then
13. \[ x_i:=(\text{CAP}−\text{SUMWEIGHT})/w_i \]
14. \[ \text{SUMWEIGHT}:=\text{SUMWEIGHT}+w_i \cdot x_i \]
15. \[ \text{SUMPROFIT}:=\text{SUMPROFIT}+p_i \cdot x_i \]
16. end

This is found in textbooks (e.g., Chapter 16.2 in [16]).

We now briefly describe the relation between fractional knapsack problem and the task assignment problem on two-type platform.

2) Relation with the task assignment problem: For a given problem instance in our scheduling problem, we can create an instance of a fractional knapsack problem as follows: (i) for each task in our scheduling problem, create a corresponding item in the fractional knapsack problem, (ii) the weight of an item in the fractional knapsack problem is the utilization of the corresponding task where the utilization here is taken for the processor on which the task executes fast and (iii) the value of an item in the fractional knapsack problem is how much lower the utilization of its corresponding task is when the task is assigned to the processor on which it executes fast as compared to its utilization if assigned to the processor on which it executes slowly. Informally speaking, we can see that if tasks could be split, then solving the fractional knapsack problem is equivalent to assigning tasks to processors so that the cumulative utilization of tasks is minimized. Again, informally speaking, we can then show that a task assignment minimizes the cumulative utilization of tasks assuming that (i) the cumulative utilization of tasks that are assigned to the processors on which they execute fast is sufficiently high and (ii) the tasks that are assigned to the processors where they execute fast has a higher ratio \( (v_i/u_i) \) than the ones that are not. Lemma 4 expressed this formally for which we provide the proof next.

3) Proof of Lemma 4: In this section, we provide a formal proof of Lemma 4. First, we (re-)introduce some of the notations. Let the task set \( \tau \) be partitioned into two disjoint subsets, \( \tau^1 \) and \( \tau^2 \). The set \( \tau^1 \) consists of those tasks which run at least as fast on a type-1 processor as on a type-2 processor; \( \tau^2 \) consists of all other tasks. In notation:

\[ \tau = \tau^1 \cup \tau^2 \]

\[ \forall \tau_i \in \tau^1: u_i \leq v_i \]

\[ \forall \tau_i \in \tau^2: u_i > v_i \]

We now (re-)state Lemma 4 and provide its proof.

**Lemma 4.** Consider \( n \) tasks and a two-type platform conforming to the system model (and notation) of Section 2. Let \( x \) denote a number such that \( 0 \leq x \leq m_1 \).

Let \( A_1 \) denote a subset of \( \tau^1 \) such that

\[ \sum_{\tau_i \in A_1} u_i > m_1 - x \]

and for every pair of tasks \( \tau_i \in A_1 \) and \( \tau_j \in \tau^1 \setminus A_1 \) it holds that \( \frac{v_i}{u_i} - 1 \geq \frac{v_j}{u_j} - 1 \). Let \( A_2 \) denote \( \tau^1 \setminus A_1 \).

Let \( B_1 \) denote a subset of \( \tau^1 \) such that

\[ \sum_{\tau_i \in B_1} u_i \leq m_1 - x \]

Let \( B_2 \) denote \( \tau \setminus B_1 \). It then holds that:

\[ \sum_{\tau_i \in A_1} u_i + \sum_{\tau_i \in A_2} v_i + \sum_{\tau_i \in \tau^2} v_i \leq \sum_{\tau_i \in B_1} u_i + \sum_{\tau_i \in B_2} v_i \]

**Proof:** Let us arbitrarily choose \( A_1, B_1 \) as defined. We will prove that this implies Inequality (54). Using Inequalities (52) and (53) we clearly get:

\[ \sum_{\tau_i \in A_1} u_i > \sum_{\tau_i \in B_1} u_i \]

With this choice of \( A_1 \) and \( B_1 \), let us consider different instances of the fractional knapsack problem:

**Instance 1:**

\[ \text{CAP} = \text{left-hand side of Inequality (55)} \]

For each \( \tau_i \in \tau \), create an item \( i \) with
$p_i = v_i - u_i$ and $w_i = u_i$

**SUMPROFIT$_1$:** value of variable SUMPROFIT when the algorithm in Lemma 6 terminates with Instance1 as input.

**Instance2:**

$\text{CAP} = \text{left-hand side of Inequality (55)}$. For each $\tau_i \in A_1$, create an item $i$ with

$p_i = v_i - u_i$ and $w_i = u_i$

SUMPROFIT$_2$: value of variable SUMPROFIT when the algorithm in Lemma 6 terminates with Instance2 as input.

**Instance3:**

$\text{CAP} = \text{right-hand side of Inequality (55)}$. For each $\tau_i \in A_1$, create an item $i$ with

$p_i = v_i - u_i$ and $w_i = u_i$

SUMPROFIT$_3$: value of variable SUMPROFIT when the algorithm in Lemma 6 terminates with Instance3 as input.

**Instance4:**

$\text{CAP} = \text{right-hand side of Inequality (55)}$. For each $\tau_i \in \tau$, create an item $i$ with

$p_i = v_i - u_i$ and $w_i = u_i$

SUMPROFIT$_4$: value of variable SUMPROFIT when the algorithm in Lemma 6 terminates with Instance4 as input.

Observe that:

**O1:** In all four instances, it holds for each element that

$\frac{p_i}{w_i} = \frac{v_i}{u_i} - 1$.

**O2:** Instance1 and Instance2 have the same capacity.

**O3:** Although Instance2 has a subset of the elements of Instance1, this subset is the subset of those elements with the largest $p_i/w_i$. (Follows from the definition of A1.)

**O4:** CAP in Instance2 is exactly the sum of the weights of the elements in A1.

**O5:** From O1-O4: SUMPROFIT$_2$=SUMPROFIT$_1$.

**O6:** Instance3 and Instance4 have the same capacity.

**O7:** Instance3 has a subset of the elements of Instance4.

**O8:** From O6 and O7: SUMPROFIT$_3$≤SUMPROFIT$_4$.

**O9:** Instance4 has smaller capacity than Instance1.

**O10:** Instance4 has the same elements as Instance1.

**O11:** From O9 and O10: SUMPROFIT$_4$≤SUMPROFIT$_1$.

**O12:** From O8 and O11: SUMPROFIT$_3$≤SUMPROFIT$_1$.

**O13:** From O12 and O5: SUMPROFIT$_3$≤SUMPROFIT$_2$.

Using O13 and the definitions of the instances and of A1 and B1 and observing that the capacity of Instance2 and Instance3 are set such that all elements in either instance will fit into the respective “knapsack”, we obtain:

$$\sum_{\tau_i \in A_1} (v_i - u_i) \leq \sum_{\tau_i \in A_1} (v_i - u_i)$$

(56)

Now, observing that $\tau = \tau^1 \cup \tau^2 = B1 \cup B2$ gives us:

$$\sum_{\tau_i \in \tau^1} v_i + \sum_{\tau_i \in \tau^2} v_i = \sum_{\tau_i \in B1} v_i + \sum_{\tau_i \in B2} v_i$$

Substituting the value of $\sum_{\tau_i \in B1} v_i$ in Inequality (56) yields:

$$\sum_{\tau_i \in \tau^1} u_i + \sum_{\tau_i \in \tau^1} v_i - \sum_{\tau_i \in \tau^2} u_i - \sum_{\tau_i \in \tau^2} v_i \leq \sum_{\tau_i \in B1} u_i + \sum_{\tau_i \in B2} u_i - \sum_{\tau_i \in B1} v_i - \sum_{\tau_i \in B2} v_i$$

Rearranging terms, we get:

$$\sum_{\tau_i \in B1} u_i + \sum_{\tau_i \in B2} u_i - \sum_{\tau_i \in B1} v_i - \sum_{\tau_i \in B2} v_i \leq \sum_{\tau_i \in B1} v_i + \sum_{\tau_i \in B2} v_i$$

Exploiting $A2 = \tau^1 \setminus A1$ yields:

$$\sum_{\tau_i \in B1} u_i + \sum_{\tau_i \in B2} u_i - \sum_{\tau_i \in B1} v_i - \sum_{\tau_i \in B2} v_i \leq \sum_{\tau_i \in A1} u_i + \sum_{\tau_i \in B2} v_i$$

This is the statement of the lemma. Hence the proof.

We now provide a detailed description of the simulation setup, experiments performed and the observed results.

**B. Experimental Setup and Results**

After studying the theoretical (worst-case) bound, i.e., the approximation ratio of our algorithm, PTAS$_{\text{NF}}$, we evaluate its average-case performance and compare it with prior state-of-the-art algorithm, PTAS$_{\text{LP}}$. For this purpose, we look at the following aspects: (i) how much faster processors our algorithm needs in practice in order to successfully partition a task set compared to PTAS$_{\text{LP}}$? and (ii) how fast our algorithm runs compared to PTAS$_{\text{LP}}$? Also, we look at (iii) how much pessimism is there in the theoretically derived performance bound of our algorithm, PTAS$_{\text{NF}}$?

In order to answer these questions, we performed two sets of experiments. In the first set of experiments, we compared the average-case performance of our algorithm, PTAS$_{\text{NF}}$, with PTAS$_{\text{LP}}$. Recall that the approximation ratio of both these algorithms depend on the value of the input parameter, $\epsilon$. Hence, we evaluated the performance of both the algorithms for different values of $\epsilon$. We observed that, in the experiments with randomly generated task sets, our algorithm requires significantly smaller processor speedup (for finding a feasible partitioning) than PTAS$_{\text{LP}}$. We also observed that our algorithm runs faster by orders of magnitude compared to PTAS$_{\text{LP}}$. Overall, in the simulations, PTAS$_{\text{NF}}$ exhibited better average-case performance by outperforming the prior state-of-the-art algorithm, PTAS$_{\text{LP}}$. In the second set of experiments, in order to see how much pessimism our theoretical analysis has, we simulated only PTAS$_{\text{NF}}$ for different values of $\epsilon$. We observed that, it performs significantly better in simulations by requiring much smaller processor speedup than indicated by its theoretical bound. We now discuss both the cases in detail.

1) **Comparison with state-of-the-art:** We implemented both the algorithms using C on Windows XP on an Intel Core2 (2.80 GHz) machine. For PTAS$_{\text{LP}}$, which relies on solving linear programming formulations, we used one of the state-of-the-art LP/ILP solvers, IBM ILOG CPLEX [17].

The algorithm, PTAS$_{\text{LP}}$, proposed in [1], for partitioning the task set on a heterogeneous multiprocessor platform, can be summarized as follows:

- The given task set is transformed into another task set by “rounding up” the utilizations to some specific values that are determined based on the value of $\epsilon$.
- The tasks in the transformed task set are grouped into big and small tasks based on their utilizations. For big tasks,
different feasible patterns are generated using dynamic programming.

- For a feasible pattern, the task assignment problem (for both big and small tasks) is formulated as an ILP and then relaxed to LP. The LP formulation is solved using an LP solver. If a feasible solution is returned by the LP solver then go to next step else consider the next feasible pattern and repeat this step.

- Using the values of the indicator variables from the solution returned by the LP solver, construct a bipartite graph and define a fractional matching. In the bipartite graph, one set of nodes represents tasks and another set of nodes represent processors. The fractional matching represents how much fraction of a task (indicated by the value of the indicator variable) is assigned to the processor to which it is connected.

- Using any maximum cardinality bipartite matching algorithm (such as Ford-Fulkerson algorithm — see pp. 714 in [16]), find an integral matching from the fractional matching. This integer matching gives the partitioning of tasks to processors.

For a given task set, a parameter $\epsilon$ and an algorithm $A$ ($A$ is either PTAS$_{NF}$ or PTAS$_{LP}$), we define the minimum required speedup factor as the minimum amount of extra speed of processors that $A$ needs, so as to succeed in finding a feasible partitioning as compared to an optimal partitioning algorithm. We denote the minimum required speedup factor of PTAS$_{NF}$ and PTAS$_{LP}$ by MRSF$_{NF}$ and MRSF$_{LP}$, respectively. For different values of $\epsilon$, for many task sets, we assess the average-case performance of both the algorithms by measuring their (i) minimum required speedup factors and (ii) running times.

The problem instances (number of tasks, their utilizations and the number of processors of each type) were generated randomly. Each problem instance had at most 10 tasks and at most 2 processors of each type. We generated 5000 task sets$,^4$ denoted as $\{\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(5000)}\}$, which we transformed into “critically feasible task sets”. We define a critically feasible task set as a task set which is partitioned feasible on a given two-type platform but rendered (partitioned) infeasible if all the task utilizations (i.e., both $u_i$ and $v_i$) are increased by an arbitrarily small factor. The intuition behind using critically feasible task sets in our simulations is that it is “hard” to find a feasible partitioning for these task sets since only a few task assignments are feasible amongst all possible assignments. Hence, by using such task sets in our simulations, we believe our evaluations have been fair and unbiased.

To obtain a partitioned critically feasible task set $\tau^{(k)}_{\text{crit}}$, from a randomly generated task set $\tau^{(k)}$, $k \in [1, 5000]$, we perform the partitioning of $\tau^{(k)}$ by formulating the problem as an Integer Linear Program as shown in Figure 3 and feeding it to a solver (such as IBM ILOG CPLEX) which outputs $Z$, the utilization of the most utilized processor. Then, we multiply all the task utilizations with $1/Z$ and repeatedly feed it back to the solver until $0.9 < Z \leq 1$, which gives $\tau^{(k)}_{\text{crit}}$.

Minimize $Z$ subject to the following constraints:

1. $\sum_{i=1}^{m} x_{ij} = 1 \quad (i = 1, 2, \ldots, n)$
2. $\sum_{i=1}^{n} (x_{ij} \cdot u_{ij}) \leq Z \quad (j = 1, 2, \ldots, m)$
3. $x_{ij}$ are non-negative integers \( (i = 1, 2, \ldots, n) \)

Fig. 3: Integer Linear Programming formulation to find a feasible partitioning of $\tau^{(k)}$ on $\pi \times x_{ij}$ are indicator variables and $u_{ij}$ are utilizations.

For a given $\epsilon$, for each critically feasible task set $\tau^{(k)}_{\text{crit}}$ and algorithm $A$ (where $A$ is either PTAS$_{NF}$ or PTAS$_{LP}$), we measure the minimum required speedup factor, denoted by MRSF$_{A}^{(\epsilon)}$. For a given $\epsilon$, Algorithm 4 shows how we compute MRSF$_{A}^{(\epsilon)}$ for every partitioned critically feasible task set, $\tau^{(k)}_{\text{crit}}$. On line 3, we initially set MRSF$_{A}^{(\epsilon)}$ to 1.0 as it denotes the speed of processors on which an optimal algorithm succeeds in finding a feasible partitioning of $\tau^{(k)}_{\text{crit}}$. Then, we input the task set to algorithm $A$ (on line 5) and if $A$ cannot find a feasible assignment, the minimum speedup factor MRSF$_{A}^{(\epsilon)}$ is incremented by a small value, here 0.01 (on line 7), and the original $u_i$ and $v_i$ of each task of $\tau^{(k)}_{\text{crit}}$ are divided by the new speedup factor (on line 8) in order to simulate the faster platform and this resulting task set is fed back to algorithm $A$. These steps (speedup factor adjustment and feeding back the derived task set) are repeated until the the algorithm $A$ succeeds in finding a feasible partitioning, which gives us the minimum required speedup factor of $A$ for the task set under consideration. This entire procedure is repeated for 5000 critically feasible task sets. Algorithm 4 is repeatedly called with different values of $\epsilon$, specifically, we used $\epsilon = 0.1, 0.2, 0.25$ and 0.3.

With this procedure, we obtain the histograms of MRSFs for both the algorithms for different values of $\epsilon$. Figure 4 shows the histograms. As can be seen from Figure 4b, in the experiments with $\epsilon = 0.2$, the MRSF$_{NF}$ never exceeded 1.12 which is 20% from the optimal value of 1.0 compared to its upper bound of $1 + 3\epsilon = 1.60$, i.e., $\frac{12-1}{0.6-1} \times 100 = 20\%$, whereas MRSF$_{LP}$ is as high as 1.30 which is 60% from the optimal value of 1.0 compared to its upper bound of $\frac{12+\epsilon}{1+\epsilon} = 1.50$, i.e., $\frac{12-1}{0.6-1} \times 100 = 60\%$. Therefore, in simulations, PTAS$_{NF}$ requires much smaller processor speedup compared to PTAS$_{LP}$ in order to find a feasible partitioning. As can be seen from Figure 4, the observations for other values of $\epsilon$ follow the same trend.

We also measure the average running times of both the algorithms for different values of $\epsilon$. In these experiments, the speedup factor is set to $1 + 3\epsilon$ for PTAS$_{NF}$ and to $\frac{1+\epsilon}{1+\epsilon}$ for PTAS$_{LP}$. This ensures that both the algorithms succeed in finding a feasible partitioning for a given task set in a single run and hence the experiments are not biased to give advantage to any of the algorithms. In our experiments with 5000 task
sets, as can be seen in Table II, for $\epsilon = 0.1$, for each task set, PTAS_{NF}, has an average running time of 128 $\mu$s, whereas the PTAS_{LP} has an average running time of 6583384 $\mu$s. Hence, for $\epsilon = 0.1$, for each task set, PTAS_{NF} runs approximately 50000 times faster compared to PTAS_{LP}. This factor is even higher for other values of $\epsilon$ as illustrated in Table II.

To summarize, in simulations, our algorithm exhibits a better average-case performance by requiring significantly smaller processor speedup for finding a feasible partitioning and by running orders of magnitude faster compared to PTAS_{LP}. Overall, PTAS_{NF} outperforms prior state-of-the-art algorithm, PTAS_{LP}.

2) Evaluation of PTAS_{NF} for different values of $\epsilon$. In order to understand how much pessimism is there in the analysis of PTAS_{NF}, we evaluated its performance for different values of $\epsilon$. In this set of experiments, we chose larger number of problem instances with each problem instance being more complex. We generated 10000 critically feasible task sets where each task set had at most 25 tasks and at most 3 processors of each

**Algorithm 4:** Pseudo-code for determining the minimum required speedup factor, $\text{MRSF}_{A}(\epsilon)$.

```
Input : Algorithm $A$ (either PTAS_{NF} or PTAS proposed in [1])
Output: The critically feasible task sets $\{\tau_{(1)}^{\text{crit}}, \tau_{(2)}^{\text{crit}}, \ldots, \tau_{(5000)}^{\text{crit}}\}$

1. step $\leftarrow 0.01$;
2. for $k = 1$ to $5000$ do
3. MRSF_{A}(k) $\leftarrow 1.0$;
4. while true do
5. result $\leftarrow$ call $A(\tau_{\text{crit}}^{(k)}, \text{assignment})$;
6. if result $\neq$ SUCCESS then
7. MRSF_{A}(k) $\leftarrow$ MRSF_{A}(k) + step;
8. $\tau_{\text{crit}}^{(k)}$ $\leftarrow$ $\tau_{\text{crit}}^{(k)}$ $\times$ ($1 / \text{MRSF}_{A}(k)$);
9. else break;
10. end
11. return $\{\text{MRSF}_{A}^{(1)}(\epsilon), \text{MRSF}_{A}^{(2)}(\epsilon), \ldots, \text{MRSF}_{A}^{(50000)}(\epsilon)\}$
```

**TABLE II:** Comparison of average running times of PTAS_{NF} and PTAS_{LP} (in $\mu$s).

<table>
<thead>
<tr>
<th>Value of $\epsilon$</th>
<th>PTAS_{NF}</th>
<th>PTAS of [1]</th>
<th>Ratio of avg. run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10</td>
<td>128.57</td>
<td>6583384.71</td>
<td>51204</td>
</tr>
<tr>
<td>0.15</td>
<td>45.33</td>
<td>6914127.72</td>
<td>152192</td>
</tr>
<tr>
<td>0.20</td>
<td>18.40</td>
<td>4449061.29</td>
<td>241796</td>
</tr>
<tr>
<td>0.25</td>
<td>10.48</td>
<td>1564960.39</td>
<td>149242</td>
</tr>
<tr>
<td>0.30</td>
<td>7.17</td>
<td>465894.09</td>
<td>64978</td>
</tr>
</tbody>
</table>

Fig. 4: Comparison of minimum required speedup factors of PTAS_{NF} and PTAS_{LP} for different values of $\epsilon$ (if an algorithm has low MRSF for many task sets then the algorithm is said to perform well).
type. Since we do not run PTAS\textsubscript{LP} (which takes much longer to output the solution as it relies on solving the linear program formulations) in this batch of experiments, we could increase the problem instances and size of each problem compared to previous set of experiments. Then, for different values of $\epsilon$, we ran PTAS\textsubscript{NF} on these 10000 critically feasible task sets and obtained histograms of MRSF\textsubscript{NF}. Figure 5 shows the histograms. As can be seen from Figure 5c, for example, in the experiments with $\epsilon = 0.3$, for almost 98% of the task sets, the MRSF\textsubscript{NF} did not exceed 1.06 which is approximately 7% of its theoretical bound (i.e., $1 + 3\epsilon = 1.90$), for the remaining 2% of the task sets, the factor did not exceed 1.12 which is approximately 13% of its theoretical bound. Thus, in the simulations, for the vast majority of task sets, our algorithm requires much smaller processor speedup than indicated by its approximation ratio. As can be seen from Figure 5, the observations for other values of $\epsilon$ follow the same trend.

Hence, PTAS\textsubscript{NF} performs significantly better in simulations than indicated by its theoretical bound.