Greedy and Dynamic Programming Algorithms for Scheduling Deadline-Sensitive Parallel Tasks

Xiaohu Wu*, Patrick Loiseau†

*EURECOM, France; Email: xiaohu.wu@eurecom.fr
†EURECOM, France; Email: patrick.loiseau@eurecom.fr

Abstract—Due to the ubiquitous batch data processing in cloud computing, the fundamental model of scheduling malleable batch tasks and its extensions have received significant attention recently. In this model, a set of $n$ tasks is to be scheduled on $C$ identical machines and each task is specified by a value, a workload, a deadline and a parallelism bound. Within the parallelism bound, the number of the machines allocated to a task can vary over time and its workload will not change accordingly. In this paper, the two core results of this paper are to quantitatively characterize a sufficient and necessary condition such that a set of malleable batch tasks with deadlines can be feasibly scheduled on $C$ machines, and to propose a polynomial time algorithm to produce such a feasible schedule. The core results provide a conceptual tool and an optimal scheduling algorithm to enable proposing new analysis and design of algorithms or improving existing algorithms for extensive scheduling objectives.

I. INTRODUCTION

Cloud computing has become the norm for a wide range of applications and batch processing platform has also become the most significant paradigm of computing workload in the cloud. With the ubiquitous batch data processing in cloud computing, many applications such as web search index update, monte carlo simulations, big-data analytics require the execution on computing clusters of batch jobs, i.e., non-real-time jobs with some flexibility in the execution period. On the one hand, while the public cloud providers typically rent virtual machines (i.e., computing power) by the hour, what really matters for tenants is completion of their jobs within a set of associated constraints (e.g., parallelism constraint, quality of service, deadline), regardless of the time of execution and computing power used. This gap between providers offer and tenants goal has received significant attention aiming to allow tenants to describe more precisely the characteristics of their jobs [1], [2] and to design appropriate mechanisms to make every tenant truthfully report the characteristics of their tasks (e.g., value) to the cloud. In the meantime, such technical progress also raises new algorithmic challenges on how to optimally schedule a type of malleable batch jobs arising in cloud computing to maximize the social welfare (i.e., the total value of tasks completed by their deadlines) [3], [4], [7], [8], [9], [11].

On the other hand, batch processing paradigms such as MapReduce and SCOPE on Dryad are being increasingly adopted in business environments and even certain real-time processing applications at Facebook and Microsoft also use the batch-processing paradigm to achieve a faster response [6], [10], [5]. As a private cloud or a tenant of the public cloud, they usually run business-critical tasks, and the organizations require to meet strict service-level agreements (SLAs) on latency, such as finishing in a certain deadline. Missing a deadline often has a significant consequence for the business (e.g., delays in updating web site content), and can further lead to financial penalties to third parties. Such context also brings new algorithmic challenges on how to optimally schedule malleable batch tasks under other time metrics, e.g., machine minimization [10], [12]. For the machine minimization, the algorithm will find the minimal amount of machines to complete a set of tasks by their respective deadlines. In fact, IBM is also integrating the smarter scheduling algorithms for various time metrics than the popular dominant resource fairness strategy into their batch processing platform for better efficiency [13].

Hence, a timely and important algorithmic challenge posed here is to further our understanding of scheduling malleable batch tasks arising in cloud computing. In this paper, we reconsider the fundamental model of [3], [4] of scheduling a set of $n$ malleable batch tasks on $C$ identical machines. All the jobs are available from the start and each of them is specified by a workload, a parallelism bound, a deadline and a value. Here, the number of machines assigned to a task can change during the execution and the parallelism bound decides the maximum amount of machines that can process a task simultaneously; however, the workload that is needed to complete a task will not change with the number of machines. In the scheduling theory, the model can be viewed as an extension of the classic model of scheduling preemptive tasks on multiple machines and arises in the ubiquitous offline batch data processing in cloud computing. Furthermore, many efforts have been devoted to the online version of the above model [7], [9], [9] and its extension [10], [11], [12] in which each task contains several subtasks with precedence constraints.

For the fundamental model in [3], [4] under the above scheduling objective, Jain et al. have proposed an $(1 - \frac{C}{2k(1-\epsilon)})$-approximation algorithm via deterministic rounding of linear program in [3] and a greedy algorithm GreedyRTL via dual fitting technique that achieves an approximation ratio of $\frac{C}{2k} + \frac{1}{k}$ in [4]. Here, $k$ is the maximal parallelism bound of tasks, and intuitively $s (\geq 1)$ is the slackness which characterizes the resource allocation flexibility (e.g., $s = 1$ means that the maximal amount of machines have to be allocated to the task at every time slot by the deadline to ensure full completion). In practice, the tasks tend to be
recurring, e.g., they are scheduled periodically on an hourly basis [10]. Hence, we can assume that the maximal deadline of tasks is finitely bounded by a constant $M$. In addition, the parallelism bound $k$ is usually a system parameter and can also viewed as a constant [13]. In this sense, the GreedyRTL algorithm has a polynomial time complexity of $O(n^2)$.

Both algorithmic design techniques in [3], [4] are based on the theory of linear program; they formulate the original problem as an integer program (IP) and relax the IP as a relaxed linear program (LP). In [3], the technique needs to solve the LP for a fractional optimal solution and manage to round the fractional solution of LP to an integer solution of IP that corresponds to an approximate solution to the original problem. In [4], the dual fitting technique needs to find the dual of the LP, and to construct a feasible solution $X$ to the dual in a greedy way. The solution to the dual corresponds to a feasible solution $Y$ to the original problem, and, due to the weak duality, the value of the dual under the solution $X$ (in the form of the social welfare under the solution $Y$ multiplied by a parameter $\alpha$) will be an upper bound of the optimal value of IP, i.e., the social welfare of an optimal solution to the original problem, where $\alpha \geq 1$. Therefore, the approximation ratio of the algorithm involved in the dual also becomes clear and is $1/\alpha$. Here, this ratio is a lower bound of the ratio of the social welfare obtained by an algorithm to the optimal social welfare.

Due to the theoretical constraints of these techniques based on LP in [3], [4], it is difficult for us to make more progress in designing better or other types of algorithms for scheduling malleable tasks with deadlines. Indeed, the design of the algorithm in [3] has to rely on the formulation of the original problem as a relaxed LP. However, for the greedy algorithm in [4], without using LP, we may have different angles than dual fitting technique to finely understand a basic question: what resource allocation features related to tasks can further benefit the performance of an algorithm that schedules this type of tasks? This question is related to the scheduling objective and the technique (e.g., greedy, dynamic programming) used to design an algorithm. Further, an answer to the following question will be proven to play a crucial role in (i) understanding the above basic question, (ii) enabling the application of dynamic programming technique to the problem in [3], [4], and (iii) algorithmic design for other time metrics such as machine minimization: how could we achieve an optimal schedule so that $C$ machines are optimally utilized by a set of malleable tasks with deadlines in terms of resource utilization? An intuitive consequence is that any algorithm for any scheduling objective would not be optimal if the machines are not optimally utilized, and its performance could be improved by optimally utilizing the machines to allow more tasks to be completed.

The importance and possible applications of an answer to the above core question may also be illustrated by its special case of scheduling on a single machine since the special case shares some common features with the general case. In this case, the famous EDF (Earliest Deadline First) rule can lead to an optimal schedule [17]. The EDF algorithm was initially designed as an exact algorithm for scheduling batch tasks to minimize the maximum job lateness (i.e., job’s completion time minus due date). So far, it has been extensively applied (i) to design exact algorithms for the extended model with release times and for scheduling with deadlines (and release times) to minimize the total weight of late jobs [16], [17], and (ii) as a significant principle in schedulability analysis for real-time systems [20].

In this paper, we will propose a new conceptual framework to address the problem of scheduling malleable batch tasks with deadlines. As discussed in the above GreedyRTL algorithm, we assume that the maximal deadline to complete a task and the maximal parallelism bound of tasks can be finitely bounded by constants and the related results of this paper are summarized as follows.

**Core Result.** The core result of this paper is the first optimal scheduling algorithm so that $C$ machines are optimally utilized by a set of malleable batch tasks $S$ with deadlines in terms of resource utilization. In this result, we understand the basic constraints of malleable tasks and identify the optimal state in which $C$ machines can be said to be optimally utilized by a set of tasks. Then, we propose a scheduling algorithm LDF($S$) that achieves such an optimal state. The LDF($S$) algorithm has a polynomial time complexity of $O(n^2)$ and is different from the EDF algorithm that corresponds to an optimal schedule in the single machine case.

**Applications.** The above core results have applications in several new or existing algorithmic design and analysis for scheduling malleable tasks under extensive objectives:

(i) an improved greedy algorithm GreedyRLM with an approximation ratio $\frac{\alpha-1}{\alpha}$ for the social welfare maximization problem with a polynomial time complexity of $O(n^2)$;
(ii) the first exact dynamic programming (DP) algorithm for the social welfare maximization problem with a pseudo-polynomial time complexity of $O(\max\{n^2, nCL(M^L)\})$;
(iii) the first exact algorithm for the machine minimization problem with a polynomial time complexity of $O(n^2)$.
(iv) an improved polynomial time approximation algorithm for the general minimax objective, reducing the approximation ratio from $(2, 1+\epsilon)$ to $(1, 1+\epsilon)$, where an algorithm is said to be an $(\alpha, \beta)$-approximation if it produces a schedule that achieves $\alpha$ times the value of an optimal solution under the original speed of machines, using the machines that are $\beta$ times the original speed.

Here, $L, D, k$ and $M$ are the number of deadlines, the maximal workload, the maximal parallelism bound, and the the bound of the maximal deadline of tasks. In addition, we also prove that $\frac{\alpha}{\alpha-1}$ is the best approximation ratio that a general greedy algorithm can achieve. Although GreedyRLM only improves GreedyRTL in [4] marginally in the case where $C \gg k$, theoretically it is the best possible. In addition, the exact algorithm for social welfare maximization can work efficiently only when $L$ is small since its time complexity is exponential in $L$. However, this may be reasonable in a machine scheduling context. In scenarios like the ones in [10], [11], the tasks are
often scheduled periodically, e.g., on an hourly or daily basis, and many tasks have a relatively soft deadline (e.g., finishing after four hours instead of three will not trigger a financial penalty). Then, the scheduler can negotiate with the tasks and select an appropriate set of deadlines \( \{ \tau_1, \tau_2, \ldots, \tau_L \} \), thereafter rounding the deadline of a task down to the closest \( \tau_i \) \( (i \in [L]) \). By reducing \( L \), this could permit to use the DP algorithm rather than GreedyRLM in the case where the slackness \( s \) is close to 1. With \( s \) close to 1, the approximation ratio of GreedyRLM approaches 0 and possibly little social welfare is obtained by adopting GreedyRLM while the DP algorithm can still obtain the maximal social welfare.

Finally, the exact algorithm for social welfare maximization can be viewed as an extension of the pseudo-polynomial time exact algorithm in the single machine case [10] that is also designed via the general dynamic programming procedure. However, before our work, we even did not know how to enable this new application and will be obtained through a complex algorithmic analysis.

The outline of this paper is as follows. In Section 2, we introduce the model of machines and tasks and the scheduling objectives considered in this paper. In Section 2, we identify what the optimal resource utilization state is and propose such a scheduling algorithm that achieves the optimal state. In Section 2, we show four applications of the results in Section 3 in different algorithmic design techniques and scheduling objectives. Finally, we make a conclusion for the paper.

II. MODEL

There are \( C \) identical machines and a set of tasks \( T = \{ T_1, T_2, \ldots, T_n \} \). The task \( T_i \) is specified by several characteristics: (1) value \( v_i \), (2) demand (or workload) \( D_i \), (3) deadline \( d_i \), and (4) parallelism bound \( k_i \). Time is discrete and we assume that the time horizon is divided into \( d \) time slots: \( [1, 2, \ldots, d] \), where \( d = \max_{T_i \in T} d_i \) and the length of each slot may be a fixed number of minutes. A task \( T_i \) can only utilize the machines located in time slot interval \( [1, d_i] \). The parallelism bound \( k_i \) imposes that, at any time slot \( t \), \( T_i \) can be executed on at most \( k_i \) machines simultaneously. The allocation of machines to a task \( T_i \) is a function \( y_i : [1, d_i] \rightarrow \{0, 1, 2, \ldots, k_i \} \), where \( y_i(t) \) is the number of machines allocated to task \( T_i \) at a time slot \( t \in [1, d_i] \). So, the model here also implied that \( D_i, d_i \in \mathbb{Z}^+ \) for all \( T_i \in T \).

The value \( v_i \) of a task \( T_i \) can be obtained only if it is fully allocated by the deadline, i.e., \( \sum_{t \leq d_i} y_i(t) \geq D_i \), and partial execution of a task yields no value. Let \( k = \max_{T_i \in T} k_i \) be the maximum parallelism bound. For the system of \( C \) machines, denote by \( W(t) = \sum_{i=1}^{m} y_i(t) \) the workload of the system at time slot \( t \) and by \( W(t) = C - W(t) \) its complementary, i.e., the amount of available machines at time \( t \). We call time \( t \) to be fully utilized (resp. saturated) if \( W(t) = 0 \) (resp. \( W(t) < k \)), and to be not fully utilized (resp. unsaturated) otherwise, i.e., if \( W(t) > 0 \) (resp. \( W(t) \geq k \)). In addition, the tasks tend to be recurring in practice, e.g., they are scheduled periodically on an hourly basis [10]. Hence, we can assume that the maximal deadline of tasks is finitely bounded by a constant \( M \). The parallelism bound \( k \) is usually a system parameter and is also assumed to be a constant [12].

Given the model above, the following scheduling objectives will be addressed separately in this paper:

- **social welfare maximization:** choose an appropriate subset \( S \subseteq T \) and produce a feasible schedule of \( S \) so as to maximize the social welfare \( \sum_{T_i \in S} v_i \) (i.e., the total value of the tasks completed by their deadlines).
- **machine minimization:** minimize the number of machines \( C \) so that there exists a feasible schedule of \( T \) on \( C \) machines.
- **the general minimax objective:** Let \( t_i \) be the completion time of a task \( T_i \) and examples of this objective include minimizing the (weighted) maximum completion time (i.e., \( \min_{T_i \in T} (v_i(t_i)) \) and the (weighted) maximum lateness of jobs (i.e., \( \min_{T_i \in T} (v_i(t_i - d_i)) \) etc..

Here, a feasible schedule means: (i) every scheduled task is fully allocated by its deadline and the constraint from the parallelism bound is not violated, and (ii) at every time slot \( t \) the number of used machines is no more than \( C \), i.e., \( W(t) \leq C \).

**Additional Notation.** We now introduce more concepts that will facilitate the subsequent algorithm analysis. We will denote by \( [l] \) and \( [l]^+ \) the sets \( \{0, 1, \ldots, l\} \) and \( \{1, 2, \ldots, l\} \). Let \( \text{len}_i = \lceil D_i/k_i \rceil \) denote the minimal length of execution time of \( T_i \). Given a set of tasks \( T \), the deadlines \( d_i \) of all tasks \( T_i \in T \) constitute a finite set \( \{ \tau_1, \tau_2, \ldots, \tau_L \} \), where \( L \leq n, \tau_1 < \tau_2 < \ldots < \tau_L = d \). Let \( D_i = \{ T_{i,1}, T_{i,2}, \ldots, T_{i,n_i} \} \) denote the set of tasks with deadline \( \tau_i \) \( (i \in [L]) \). Let \( D_{i,j} \) denote the set of tasks with deadline \( \tau_i \) and the minimal length of execution time in \( (\tau_i - \tau_{i-j+1}, \tau_i - \tau_{i-j}) \). Denote by \( s_i = \frac{d_i}{\text{len}_i} \) the slackness of the least flexible task \( T_i \), measuring the time flexibility of machine allocation (e.g., \( s_i = 1 \) may mean that \( T_i \) should be allocated the maximal amount of machines \( k_i \) at every \( t \in [1, d_i] \) and let \( s = \min_{T_i \in T} s_i \) be the slackness of the least flexible task (\( s \geq 1 \)). Denote by \( y_i^* = \frac{v_i}{s_i} \) the marginal value, i.e., the value obtained by the system through executing per unit of demand of the task \( T_i \). Finally, we assume that the demand of each task is an integer. For a set of tasks \( S \), we use its capital \( S \) to denote the total demand of the tasks in \( S \). Let \( D = \max_{T_i \in T} D_i \) be the demand of the largest task.
Throughout the paper, the details of omitted proofs can be found in the appendix or our technical report [15].

III. Optimal Schedule

In this section, we identify the optimal utilization state of C machines on which a set of tasks T is scheduled; in the meantime, we propose a scheduling algorithm that can achieve such optimal state.

A. Optimal Resource Utilization State

For the tasks in our model, the deadline dᵢ decides the time interval in which a task can utilize the machines, and the parallelism bound kᵢ restricts that Tᵢ can utilize at most kᵢ machines at every time slot in [l, dᵢ]. Let S ⊆ T, and denote Sᵢ = S ∩ Dᵢ and Sᵢ,j = S ∩ Dᵢ,j (i ∈ [L⁺], j ∈ [i⁺]). Let λᵢ(S) = ∑₁⁻⁺L⁺=m+1{∑₁⁻⁺Li,j Sᵢ,j + ∑₁⁻⁺Li=⁻⁺L⁺+1∑₁⁻⁺Jₐ∈Sᵢ,j kᵢ(τᵢ - τᵢ-m)}. Such state also ensures that when the next task Tⱼ is considered, Allocate-B(j) is able to fully allocate Dⱼ resources to it iff S′ ∪ {Tⱼ} satisfies the boundary condition. If so, LDF(S) will give a feasible schedule for a set of tasks S only if S satisfies the boundary condition.

To realize the function of Allocate-B(i) above, the cooperation among three algorithms are needed: Fully-Utilize(i), Fully-Allocate(i), and AllocateRLM(i, η₁) that are respectively presented as Algorithm 3, Algorithm 5, and Algorithm 6. Now, we introduce their executing process.

Fully-Utilize(i). Fully-Utilize(i) aims to ensure a task Tᵢ to fully utilize the current available machines at the time slots closest to its deadline with the constraint of parallelism bound. During its execution, the allocation to Tᵢ at every time slot t is done from the deadline towards earlier time slots, and Tᵢ is allocated min{ₖᵢ, Dᵢ - ∑ₜ₋₁ₜ⁺ yi(t), W(t)} machines at t. min{ₖᵢ, Dᵢ - ∑ₜ₋₁ₜ⁺ yi(t), W(t)} is the maximal amount of machines it can or need to utilize at t.

B. Scheduling Algorithm

In this section, we introduce the proposed optimal scheduling algorithm LDF(S) (last deadline first), presented as Algorithm 1.

Algorithm 1: LDF(S)

Output: A feasible allocation of machines to a set of tasks S

1. for m ← L to 1 do
2. while Sᵢ,m ≠ 0 do
3. Get Tᵢ from Sᵢ,m
4. Allocate-B(i)
5. Sᵢ,m ← Sᵢ,m - {Tᵢ}

Algorithm 2: Allocate-B(i)

1. Fully-Utilize(i)
2. Fully-Allocate(i)
3. AllocateRLM(i, 1)

Algorithm 3: Fully-Utilize(i)

Algorithm 4: Fully-Allocate(i)

Algorithm 5: AllocateRLM(i, η₁)
than but closest to \( t \) such that \( W(t') > 0 \), and Routine() checks the current condition to decide whether to exit the loop and itself or to take the subsequent operation. In particular, when \( \eta_1 = 1 \) and \( \eta_2 = 0 \), it exits the loop whenever either of the following conditions is satisfied: (i) the number of current available machines \( W(t) \) is \( \Delta \), and (ii) there exists no such \( t' \); when \( \eta_1 = 1 \) and \( \eta_2 = 1 \), the loop stops whenever either of the above two conditions is satisfied or there exists such \( t' \) but \( \sum_{i=1}^{t-1} y_i(t') \leq W(t) \). Regardless of Fully-Allocate(\( i \)) or AllocateRLM(\( i, 1 \)), if none of the corresponding exit conditions above is satisfied, there exists a task \( T_i \) such that \( y_i(t) > y_i(t') \). The existence of such \( T_i \) will be explained when we introduce Fully-Utilize(\( i \)) and AllocateRLM(\( i, \eta_1 \)). Then, it decreases the allocation \( y_i(t) \) of \( T_i \) at \( t \) by 1 and increases its allocation \( y_i(t') \) at \( t' \) by 1. This operation does not change the total allocation to \( T_i \), and violate the parallelism bound \( k_i \) of \( T_i \), since the current \( y_i(t') \) is no more than the initial \( y_i(t) \). Upon completion of the above operation, the next loop iteration begins.

Fully-Allocate(\( i \)). Fully-Allocate(\( i \)) ensures that \( T_i \) is fully allocated. Upon completion of Fully-Utilize(\( i \)), let \( \Omega = D_i - \sum_{t=1}^{a_{i+1}} y_i(t) \) denote the partial demand of \( T_i \) that remains to be allocated more resources for full completion of \( T_i \). Then, there is a loop in Fully-Allocate(\( i \)) in which time slots \( t \) are considered one by one from the deadline towards earlier time slots. For every \( t \), it checks whether or not \( \Omega > 0 \) and \( T_i \) can be allocated more machines at this time slot, namely, \( k_i - y_i(t) > 0 \). Then, let \( \Delta = \min\{k_i - y_i(t), \Omega\} \) and if \( \Delta > 0 \) it attempts to make the number of available machines at \( t \) become \( \Delta \) by calling Routine(\( \Delta, 1, 0 \)). Subsequently, the algorithm updates \( \Omega \) to be \( \Omega - W(t) \), and allocates the current available machines \( W(t) \) at \( t \) to \( T_i \). Here, upon completion of its loop iteration of Fully-Allocate(\( i \)) at \( t \), \( W(t) = 0 \) if Fully-Allocate(\( i \)) has increased the allocation of \( T_i \) at \( t \); in this case we will also see that \( W(t) = 0 \) just before the execution of this loop iteration at \( t \). Then, Fully-Allocate(\( i \)) begins its next loop iteration at \( t - 1 \).

**Lemma 3.4:** Fully-Allocate(\( i \)) will never decrease the allocation \( y_i(t) \) of \( T_i \) at every time slot \( t \) done by Fully-Utilize(\( i \)). If \( W(t) > 0 \) upon its completion, we also have that \( W(t) > 0 \) just before the execution of every loop iteration of Fully-Allocate(\( i \)).

According to Lemmas 3.3 and 3.4, we make the following observation. At the beginning of every loop iteration of Fully-Allocate(\( i \)), if \( \Delta > 0 \), we have that \( W(t) = 0 \) since the current allocation of \( T_i \) at \( t \) is still the one done by Fully-Utilize(\( i \)) and \( \Omega > 0 \); otherwise, it should have been allocated some more machines at \( t \). If there exists a \( t' \) such that \( W(t') > 0 \) in the loop of Routine(\( \eta_1 \)), since the allocation of \( T_i \) at \( t' \) now is still the one done by Fully-Utilize(\( i \)) and \( \Omega > 0 \), we can know that \( y_i(t') = k_i \). Then, we have that \( W(t) - y_i(t') > W(t') \). If there exists a task \( T_i \) such that \( y_i(t') < y_i(t) \); otherwise, we will not have that inequality. In the subsequent execution of the loop of Routine(\( \eta_1 \)), \( W(t) \) becomes greater than 0 but \( W(t) < \Delta \) for all \( W(t') < \Delta \). We still have \( W(t) - y_i(t') = C - \sum_{i=1}^{t-1} y_i(t') \) and such \( T_i \) can still be found.

AllocateRLM(\( i, \eta_1 \)). Without changing the total allocation of \( T_i \) in \([1, d_i]\), AllocateRLM(\( i, \eta_1 \)) takes the responsibility to make the time slots closest to \( d_i \) fully utilized by \( T_i \) and the other fully allocated tasks with the constraint of parallelism bound, namely, the Right time slots being Loaded Most.

To that end, there is also a loop in AllocateRLM(\( i, \eta_1 \)) that considers every time slot \( t \) from the deadline of \( T_i \) towards earlier time slots. For the current \( t \) being considered, if the total allocation \( \sum_{i=1}^{t-1} y_i(t) \) of \( T_i \) in \([1, t-1]\) is greater than 0, AllocateRLM(\( i \)) begins its loop iteration at \( t \). Let \( \Delta = \min\{k_i - y_i(t), \sum_{i=1}^{t-1} y_i(t)\} \) and \( \Delta \) is the maximal extra machines that \( T_i \) can utilize at \( t \). If \( \Delta > 0 \), we enter Routine(\( \Delta, \eta_1, \eta_2 \)). Here, \( \Delta > 0 \) also means \( y_i(t) < k_i \). When Routine() stops, we have that the number of available machines \( W(t) \) at \( t \) is no more than \( \Delta \). Let \( u_t \) be the last time slot \( t' \) considered in the loop of Routine() for \( t \) such that the total allocation at \( t' \) has been increased. In a different case than the current state here, AllocateRLM(\( i \)) does nothing and take no effect on the allocation of \( T_i \) at \( t \); then set \( u_t = u_{t+1} \).
Algorithm 6: AllocateRLM(i, η1)

1. \( t \leftarrow d_i \)
2. while \( \sum_{t'=1}^{t-1} y_{i}(t') > 0 \) do
3. \( \Delta \leftarrow \min\{k_i - y_{i}(t), \sum_{t'=1}^{t-1} y_{i}(t')\} \)
4. Routine(Δ, η1, 1)
5. \( \theta \leftarrow \overline{W}(t), y_i(t) \leftarrow y_i(t) + \overline{W}(t) \)
6. let \( t'' \) be such a time slot that \( \sum_{t'=1}^{t''} y_{i}(t') < \theta \) and \( \sum_{t'=1}^{t''} y_{i}(t') \geq \theta \)
7. \( \theta \leftarrow \theta - \sum_{t'=1}^{t''} y_{i}(t'), y_i(t'') \leftarrow y_i(t'') - \theta \)
8. for \( \bar{t} \leftarrow 1 \) to \( t'' - 1 \) do
9. \( y_i(\bar{t}) \leftarrow 0 \)
10. \( t \leftarrow t - 1 \)

if \( t < d_i \) and \( u_\ell = d_\ell \) if \( t = d_\ell \). Then, AllocateRLM(i, η1) decreases the current allocation of \( T_\ell \) at the earliest time slots in \([1, u_\ell - 1] \) to 0 (and by \( \overline{W}(t) \)), and accordingly increases the allocation of \( T_\ell \) at \( t \). Here, upon completion of the loop iteration of AllocateRLM(\( i \)) at \( t \), \( \overline{W}(t) \) also equals to 0 just before the execution of this loop iteration at \( t \). Here, when AllocateRLM(\( i \)) is called in Allocate-B(\( i \)), \( \eta_1 = \eta_2 = 1 \). The reason for the existence of \( T_\ell \) is similar to but more complex than the case of Fully-Allocate(\( i \)), and is explained.

**Difference of Routine(\( i \)) and GreedyRTL.** The operations in Routine(\( i \)) are the same as the ones in the inner loop of AllocateRTL(\( i \)) in GreedyRTL [4] and the differences are the exit conditions of the loop. In AllocateRTL(\( i \)), one exit condition is that there is no unsaturated time slot \( t'' \) earlier than \( t \). In this case, although GreedyRTL can guarantee the optimal resource utilization in a particular time interval [15] according the state we identified in Section III-A, there inevitably exist unsaturated time slots that are not optimally utilized. In fact, by our analysis in [15], GreedyRTL achieves a resource utilization of \( \min\{ \frac{\overline{W}(t)}{c_{i+1}}, C_{i+1}\} \) due to its allocation condition, which is not optimal.

**Proposition 3.1:** The boundary condition is sufficient for LDF(\( S \)) to produce a feasible schedule for a set of malleable tasks with deadlines \( S \). The time complexity of LDF(\( S \)) is \( O(n^2) \).

By Proposition 3.1 and Lemma 3.2 we have the following theorem:

**Theorem 3.1:** A feasible schedule for a set of tasks \( S \) can be constructed on \( C \) machines if and only if the boundary condition holds.

In other words, if LDF(\( S \)) cannot produce a feasible schedule for a set of tasks \( S \), then this set cannot be successfully scheduled by any algorithm.

IV. APPLICATIONS

In this section, we show the applications of the results in Section III to two algorithmic design techniques for the problem in [8], [9], giving the best possible greedy algorithm and the first exact dynamic programming algorithm. We also show its direct applications to the machine minimization problem and the scheduling problem with the general minimax objectives.

A. Greedy Algorithm

In this section, we give a relatively thorough exploration to the application of greedy algorithm in the social welfare maximization problem. The concept of optimal resource utilization state in Section III-A plays a core role in our new analysis of a generic greedy algorithm, revealing the key factors deciding its performance guarantee. In the meantime, we also give the best possible performance guarantee a general greedy algorithm can achieve. The optimal schedule in Section III-A finally enables us to propose the best possible greedy algorithm.

**Generic algorithm.** Greedy algorithms are often the first algorithms one considers for many optimization problems. In
terms of the maximization problem, the general form of a greedy algorithm is as follows \cite{19, 18}: it tries to build a solution by iteratively executing the following steps until no item remains to be considered in a set of items: (1) selection standard: in a greedy way, choose and consider an item that is locally optimal according to a simple criterion at the current stage; (2) feasibility condition: for the item being considered, accept it if it satisfies a certain condition such that this item constitutes a feasible solution together with the tasks that have been accepted so far under the constraints of this problem, and reject it otherwise. Here, an item that has been considered and rejected will never be considered again. The selection criterion is related to the objective function and constraints, and is usually the ratio of ‘advantage’ to ‘cost’, measuring the efficiency of an item. In the problem of this paper, the constraint comes from the capacity to hold the chosen tasks and the objective is to maximize the social welfare; therefore, the selection criterion here is the ratio of the value of a task to its demand.

Given the general form of greedy algorithm, we define a class \textsc{Greedy} of algorithms that operate as follows: (i) considers tasks in the non-increasing order of the marginal value; and (ii) let \(A\) denote the set of the tasks that have been accepted so far, and, for a task \(T_i\) being considered, it is accepted and fully allocated \(\text{iff}\) there exists a feasible schedule for \(\mathcal{A} \cup \{T_i\}\). In the following, we refer to the generic algorithm in \textsc{Greedy} as \textsc{Greedy}.

**Proposition 4.1:** The best performance guarantee that a greedy algorithm can achieve is \(\frac{\Delta}{2}\).

**Notation.** To describe the resource allocation process of a greedy algorithm, we define the sets of consecutive accepted (i.e., fully allocated) and rejected tasks \(\mathcal{A}_1, \mathcal{R}_1, \mathcal{A}_2, \cdots\). Specifically, let \(\mathcal{A}_m = \{T_{i_m}, T_{i_m+1}, \cdots, T_{j_m-1}\}\) be the \(m\)-th set of all the adjacent tasks that are fully allocated after the task \(T_{j_m}\), where \(T_{j_m}\) is the first rejected task following the set \(\mathcal{A}_m\). Correspondingly, \(\mathcal{R}_m = \{T_{j_m}, \cdots, T_{i_{m+1}-1}\}\) is the \(m\)-th set of all the adjacent rejected tasks following the set \(\mathcal{A}_m\), where \(m \in [K]^+\) for some integer \(K\) and \(i_1 = 1\). Integer \(K\) represents the last step: in the \(K\)-th step, \(\mathcal{A}_L \neq \emptyset\) and \(\mathcal{R}_K\) can be empty or non-empty. We also define \(c_m = \max_{T_i \in \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_m} \{d_i\}\) and \(c'_m = \max_{T_i \in \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_m} \{d_i\}\). In the following, we refer to this generic greedy algorithm as \textsc{Greedy}. While the tasks in \(\mathcal{A}_m \cup \mathcal{R}_m\) are being considered, we refer to \textsc{Greedy} as being in the \(m\)-th phase. Before the execution of \textsc{Greedy}, we refer to it as being in the 0th phase.

In the \(m\)-th phase, upon completion of the resource allocation to a task \(T_i \in \mathcal{A}_m \cup \mathcal{R}_m\), we define \(D_{i, t_{i}} = \sum_{t_{i}=t}^{t_{i+1}} y(t)\) to describe the current total allocation to \(T_i\) over \([t_1, t_2]\). After the completion of \textsc{Greedy}, we also define \(D_{K+1, t_{i}} = \sum_{t_{i}=t}^{t_{i+1}} y(t)\) to describe the final total allocation to \(T_i\) over \([t_1, t_2]\). We further define \(T_{K+1, i}\) as an imaginary task with characteristics \(\{v_{K+1, i}, D_{K+1, i}, D_{K+1, i}, k_i\}\), where \(v_{K+1, i} = t_i D_{K+1, i}/D_i\), \(D_{K+1, i} = \min \{t_2, d_i\}\). 

**Features and Theorem.** We now define two features of the resource allocation structure related to the accepted tasks. In the next section, we will show that if \textsc{Greedy} can achieve a resource allocation structure satisfying those two features, its performance guarantee can be deduced immediately.

Upon completion of the \(m\)-th phase of \textsc{Greedy}, we define the threshold parameter \(t_m^m\) as follows: If \(c_m \geq c'_m\), then set \(t_m^m = c_m\). If \(c_m < c'_m\), then set \(t_m^m\) to a certain time slot in \([c_m, c'_m]\). We emphasize here that \(d_i \leq t_m^m\) for all \(T_i \in \bigcup_{j=1}^{m} \mathcal{R}_j\) hence the allocation to the tasks of \(\bigcup_{j=1}^{m} \mathcal{R}_j\) in \([t_m^m + 1, T]\) is ineffective and yields no value due to the constraint from the deadline. For ease of exposition, we let \(t_0^m = 0\) and \(t_K^K = T\). With this notation, we define the following two features that we will want the resource allocation to satisfy for all \(m \in [K]^+\):

**Feature 4.1:** The resource utilization achieved by the set of tasks \(\bigcup_{j=1}^{m} \mathcal{A}_j\) in \([1, t_m^m]\) is at least \(r\), i.e., \(\sum_{T_i \in \mathcal{A}_m} D_{K+1, i} (C, t_m^m) \geq r\).

Viewing \(T_{K+1, i}\) as a real task with the same allocation done by \textsc{Greedy} as that of \(T_i\) in \([1, t_m^m]\), we define the second feature as:

**Feature 4.2:** \([t_m^m + 1, t_m^{m+1}]\) is optimally utilized by \(\{T_{K+1, i}, T_i \in \mathcal{A}_m\}\).

**Theorem 4.1:** If \textsc{Greedy} achieves a resource allocation structure that satisfies Feature 4.1 and Feature 4.2, it gives an \(r\)-approximation to the optimal social welfare.

For ease of the subsequent exposition, we add a dummy time slot 0 but the task \(T_i \in \mathcal{T}\) can not get any resource there, that is, \(y_i(0) = 0\) forever. We also let \(A_0 = \mathcal{R}_0 = \mathcal{A}_{K+1} = \mathcal{R}_K = \emptyset\).

**Best possible greedy algorithm.** We now introduce the executing process of the greedy algorithm \textsc{GreedyRLM} presented as Algorithm 7.

(1) considers the tasks in the non-increasing order of the marginal value.

(2) in the \(m\)-th phase, for a task \(T_i\) being considered, if it satisfies the allocation condition \(\sum_{t_i \leq t} \min \{\text{WR}(t), k_i\} \geq D_i\), call Allocate-A(\(i\)) to make \(T_i\) fully allocated. Here, Routine(\(\Delta, 0, 1\)) exits only if in a loop iteration one of the following conditions is satisfied: (1) the number of current available machines \(\text{WR}(t)\) is \(\Delta\), (2) there exists no such \(t'\), and (3) there exists such \(t'\) but either \(\sum_{t_i \leq t_i} y_i(T) \leq \text{WR}(t)\) or \(t' \leq t_m^m\). The existence of \(T_{i'}\) is also explained in the appendix.

(3) if the allocation condition is not satisfied, set the threshold parameter \(t_m^m\) of the \(m\)-th phase in the way defined by lines 8–15 of Algorithm 7.

**Proposition 4.2:** \textsc{GreedyRLM} gives an \(\frac{\Delta}{2}\)-approximation to the optimal social welfare with a time complexity of \(O(n^2)\).

**B. Dynamic Programming Algorithm**

In this section, we show how the concept of the optimal resource utilization state and the optimal schedule in Section 11 enable the design of an exact dynamic programming algorithm for the social welfare maximization problem. The proposed
algorithm has a pseudo-polynomial time complexity but may make sense in practice.

For any solution, there must exist a feasible schedule for the tasks selected to be fully allocated by this solution. So, the set of tasks in an optimal solution satisfies the boundary condition by Lemma 3.2. Then, to find the optimal solution, we only need address the following problem: if we are given $C$ machines, how can we choose a subset $S$ of tasks in $D_1 ∪ · · · ∪ D_L$ such that (i) this subset satisfies the boundary condition, and (ii) no other subset of selected tasks achieve a better social welfare? This problem can be solved via dynamic programming (DP).

To propose a DP algorithm, we need to identify a dominant condition for the model of this paper [21]. Let $F ⊆ T$ and we define a $L$-dimensional vector

$$H(F) = (λ_1^F(F) − λ_1^0(F), · · · , λ_L^F(F) − λ_L^{m−1}(F)),$$

where $λ_m^F(F) − λ_m^{m−1}(F)$, $m ∈ [L]^+$, denotes the optimal resource that $F$ can utilize on $C$ machines in the segmented timescale $[τ_L−m + 1, τ_L−m+1]$ after $F$ has utilized $λ_m^{m−1}(F)$ resource in $[τ_L−m+1, τ_L]$. Let $v(F)$ denote the total value of the tasks in $F$ and then we introduce the notion of one pair $(F, v(F))$ dominating another $(F′, v(F′))$ if $H(F) = H(F′)$ and $v(F) ≥ v(F′)$, that is, the solution to our problem indicated by $(F, v(F))$ uses the same amount of resources as $(F′, v(F′))$, but obtains at least as much value. We now give the general DP procedure DP(T) [21], presented as Algorithm 3 in the appendix. Here, we iteratively construct the lists $A(j)$ for all $j ∈ [n]^+$. Each $A(j)$ is a list of pairs $(F, v(F))$, in which $F$ is a subset of $T$, $v(F)$ is the total value of the tasks in $F$. Each list only maintains all the dominant pairs. Specifically, we start with $A(1) = \{(0, 0), (T_1, v_1)\}$. For each $j = 2, · · · , n$, we first set $A(j) ← A(j − 1)$, and then for each $(F, v(F)) ∈ A(j − 1)$, we add $(F ∪ T_j, v(F ∪ T_j))$ to the list $A(j)$ if $F ∪ T_j$ satisfies the boundary condition. We finally remove from $A(j)$ all the dominated pairs. DP(T) will select a subset $S$ of $T$ from all pairs $(F, v(F)) ∈ A(n)$ so that $v(S)$ is maximal.

Proposition 4.3: Given the subset $S$ output by DP(T), LDF(S) gives an optimal solution to the welfare maximization problem with a time complexity $O(\max\{n d^L C^L, n^2\})$.

Discussion. As in the knapsack problem [21], to construct the algorithm DP(T), the pairs of the possible state of resource utilization and the corresponding best social welfare have to be maintained and a $L$-dimensional vector has to be defined to indicate the resource utilization state. This seems to imply that we cannot make the time complexity of a DP algorithm polynomial in $L$.

C. Machine Minimization

In this section, we consider the machine minimization problem. For a set of tasks $T$, the minimal number of machines needed to produce a feasible schedule of $T$ is exactly the minimal value of $C$ such that the boundary condition is satisfied. Then, through binary search to obtain the minimal $C$ such that the boundary condition is satisfied, we have the following proposition by Proposition 5.1 and Lemma 5.2.

Proposition 4.4: There exists an exact algorithm for the machine minimization problem with a time complexity of $O(n^2)$.

D. General minimax Objective

In this subsection, we show that under the model of this paper a direction application of the theory in Section III improves the approximation ratio $α$ of the algorithm in [12] for the general minimax objective by a factor 2.

In [12], the problem of scheduling malleable tasks with precedence constraints is considered and Nagarajan et al. give a $(2, 1 + ε)$-bicriteria approximation algorithm for arbitrary minimax cost metrics, where $ε ∈ (0, 1)$ can be arbitrarily small. Here, an algorithm is said to be an $(α, β)$-approximation if it produces a schedule that achieves $α$ times the value of the optimal schedule under the original speed of machines, using the machines that are $β$ times the original speed. The minimax objective includes minimizing the (weighted) maximum completion time and the (weighted) maximum lateness of jobs etc.

For the general minimax objective, Nagarajan et al. [12] use the binary search to find a bound $V$ that is at most $(1 + ε)$ times the optimal value and then obtain the corresponding maximal deadline $d_i' = \arg \max_t \{t : v_i(t) ≤ V\}$, where $v_i(t)$ is the value obtained by a task $T_i ∈ T$ when it is completed at time $t$. In the case of minimizing the (weighted) maximum completion time, $v_i(t) = tv_i$ where $tv_i$ is the weight; in the case of minimizing the (weighted) maximum lateness of jobs, $v_i(t) = (t − d_i)v_i$. Then, Nagarajan et al. also propose an $(1, 2)$-approximation algorithm for scheduling malleable tasks with deadlines and precedence constraints only if there exists a feasible schedule for the tasks that does not violate their respective deadlines. Finally, they obtained an $(2, 1 + ε)$-approximation algorithm for arbitrary minimax cost metrics. Under the model of this paper, after finding a bound $V$ that is at most $(1 + ε)$ times the optimal value and obtaining the corresponding maximal deadline $d_i' = \arg \max_t \{t : v_i(t) ≤ V\}$ for every task, the optimal schedule in Section III can give an $(1, 1 + ε)$ approximation algorithm and we therefore have that

Proposition 4.5: There is a polynomial time $(1, 1 + ε)$-approximation algorithm for scheduling independent malleable tasks under the general minimax objective.

V. Conclusion

In this paper, we consider the problem of scheduling $n$ deadline-sensitive malleable batch jobs on $C$ identical machines. Our core result is a new theory to give the first optimal scheduling algorithm so that $C$ machines can be optimally utilized by a set of batch tasks. We further derive three algorithmic results in obvious or non-obvious ways: (i) the best possible greedy algorithm for social welfare maximization with a polynomial time complexity of $O(n^2)$ that achieves an approximation ratio of $\frac{2}{1 + \epsilon}$, (ii) the first dynamic programming
algorithm for social welfare maximization with a polynomial time complexity of $O(\max\{nd^2C^2, n^2\})$, (iii) the first exact algorithm for machine minimization with a polynomial time complexity of $O(n^2)$, and (iv) an improved polynomial time approximation algorithm for the general minimax objective, reducing the approximation ratio from $(2, 1 + \epsilon)$ to $(1, 1 + \epsilon)$, where an algorithm is said to be an $(\alpha, \beta)$-approximation if it produces a schedule that achieves $\alpha$ times the value of an optimal schedule under the original speed of machines, using the machines that are $\beta$ times the original speed. Here, $L$ and $d$ are the number of deadlines and the maximal deadline of tasks.

Future work includes exploring the possibility of extending the definition in this paper of the optimal state of executing tasks and to propose similar algorithms in this paper for those on this, one may attempt to find the optimal schedule for those of several subtasks with precedence constraints. Then, based on this, one may attempt to find the optimal schedule for those cases and to propose similar algorithms in this paper for those extended cases.

REFERENCES


APPENDIX

A. Preliminaries

This section is supplementary to the related algorithms that will be further called in LDF(\mathcal{S}) and GreedyRLM. Here, we will explain their functions and executing processes and give some lemmas. Furthermore, we will show the reason why these algorithms are feasible and produce a schedule without violating the constraints from the capacity, deadline and parallelism bound.

1) Fully-Utilize(i): Fully-Utilize(i) is presented as Algorithm 5

Proof: (of Lemma 3.3) By contradiction. If $T_i$ is not allocated $\mathcal{min}\{k_i, D_i - \sum_{t=t_i+1}^{d_i} y_i(t)\}$ machines at $t$ with $W(t) > 0$, it should have utilized some more $\mathcal{min}\{\mathcal{min}\{k_i, D_i - \sum_{t=t_i+1}^{d_i} y_i(t)\} - y_i(t), W(t)\}$ machines when being allocated at $t$. Further, if we also have $D_i - \sum_{t=t_i}^{d_i} y_i(t) > 0$, and $T_i$ is not allocated $k_i$ machines at $t$, $T_i$ should also have been allocated some more $\mathcal{min}\{D_i - \sum_{t=t_i}^{d_i} y_i(t), W(t), k_i - y_i(t)\}$ machines at $t$.

2) Fully-Allocate(i): Fully-Allocate(i) is presented as Algorithm 5

Proof: (of Lemma 3.4) The only operations of changing the allocation of tasks occur in line 5 of Fully-Allocate(i) and line 17 of Routine(\cdot). In those processes, there is no operation that will decrease the allocation of $T_i$. In line 17 of Routine(\cdot), the allocation to $T_i$ at $t'$ will be increased and the allocation to $T_i$ at $t$ is reduced. $W(t')$ is increased and $W(t)$ is reduced. However, in line 5 of Fully-Allocate(\cdot), $W(t)$ will becomes zero and $W(t) = C$. Hence, the allocation $W(t)$ at $t$ is also not decreased upon completion of a loop iteration of Fully-Allocate(\cdot). The lemma holds.

Let $\omega$ denote the last time slot in which Fully-Allocate(i) will increase the allocation of $T_i$. In other words, the allocation
of \( T_i \) at every time slot in \([1, \omega - 1]\) is still the one achieved by Fully-Utilize(i).

**Lemma A.1:** Upon completion of Fully-Allocate(i), \( y_i(t) = k_i \) for all \( t \in [\omega + 1, d_i] \).

**Proof:** Suppose there exists a time slot \( t \in [\omega + 1, d_i] \) at which the allocation of \( T_i \) is less than \( k_i \). By the definition of \( \omega \), we have that \( \Omega = d_i - \sum_{j=1}^{i-1} y_i(t) > 0 \) and there also exists a time slot \( t' \in [1, \omega - 1] \) that is not fully utilized upon completion of the loop iteration of Fully-Allocate(i) at \( t \). This contradicts with the exit condition of Routine(\( t \)) described in Section III and Section IV-A or in Algorithm 4. The lemma holds. \( \square \)

3) **AllocateRLM(i, \( \eta_1 \)):** AllocateRLM(i, \( \eta_1 \)) is presented as Algorithm 6. We now show some lemmas to help us identify the resource allocation state. This will further show the existence of \( T_v \) in line 16 of Routine(\( t \)) as we define a parameter. Upon completion of the loop iteration in AllocateRLM(\( t \)) at \( t \), if the allocation of \( T_i \) at \( t \) has never been changed, let \( v_i \) denote the time slot \( t'' \) in line 6 of AllocateRLM(\( t \)), where \( \sum_{t=1}^{v_i} y_i(t) = 0 \) and \( v_i \) is the farthest time slot in which the allocation of \( T_i \) is decreased. If \( y_i(t) \) is not changed by AllocateRLM(\( t \)), let \( v_i = v_i + 1 \) if \( t < d_i \) and \( v_i = 1 \) if \( t = d_i \).

**Lemma A.2:** Upon completion of the loop iteration of AllocateRLM(\( t \)) at \( t \), the allocation at every time slot in \([v_i + 1, d] \) is never decreased since the execution of AllocateRLM(\( t \)).

**Proof:** The only operations of decreasing the allocation of tasks occur in lines 6-10 of AllocateRLM(\( t \)) and line 17 of Routine(\( t \)). The operations in lines 6-10 of AllocateRLM(\( t \)) does not affect the allocation in \([v_i + 1, d] \) by the definition of \( v_i \). Although the allocation at \( t \) is decreased in line 17 of Routine(\( t \)), it will finally become \( C \) in line 5 of AllocateRLM(\( t \)). \( \square \)

**Lemma A.3:** While the loop iteration in AllocateRLM(\( t \)) for \( t \) begins until its completion, if there exists a time slot \( t'' \) in \([v_i + 1, t - 1]\) such that \( \overline{W}(t'') > 0 \), we have that

1. \( v_i t_{d_i} \leq \cdots \leq v_i < u_i \leq \cdots \leq u_d \).
2. \( t'' \) has never been utilized since the execution of AllocateRLM(\( t \)).
3. the allocation of \( T_i \) at every time slot in \([1, t - 1]\) has never been increased since the execution of AllocateRLM(\( t \)).

**Proof:** By the definition of \( v_i \) and the way that AllocateRLM(\( t \)) decreases the allocation of \( T_i \) (lines 6-10), we have \( v_i t_{d_i} \leq \cdots \leq v_i \). Further, according to the stop condition 3 of the loop of Routine(\( t \)), we always have that \( \sum_{t=1}^{v_i - 1} y_i(t) \leq \overline{W}(t) \) and further conclude that \( v_i < u_i \) since the allocation at \( t \) can not be decreased by lines 6-10 of AllocateRLM(\( t \)). The allocation at \( t'' \) in \([v_i + 1, t - 1]\) is not decreased by AllocateRLM(\( t \)) at any moment since \( v_{d_i} \leq \cdots \leq v_i < t'' \) and \( t'' < t \). Lemma A.3(2) holds. By Lemma A.3(2), the \( t'' \) in the loop iteration for \( t \) is also not fully utilized in the previous loop iteration. Hence, \( u_i \leq u_{t''} \) for a \( t'' \in [t + 1, d_i] \). Lemma A.3(1) holds. So far, AllocateRLM(\( t \)) only attempts to increase the allocation of \( T_i \) in \([t, d_i] \) and there is no operation to increase its allocation in \([1, t - 1]\). Lemma A.3(3) holds. \( \square \)

The lemma below follows directly from Lemma A.3(1):

**Lemma A.4:** Upon completion of AllocateRLM(\( t \)), we have that \( v_i t_{d_i} \leq \cdots \leq v_{d_i} \leq v_i < u_{d_i} \leq u_i \leq u_d \).

We next show the existence of \( T_v \) in line 16 of Routine(\( t \)) as we define a parameter. Upon completion of AllocateRLM(\( t \)) and lines 6-10 of AllocateRLM(\( t \)) if there exists a time slot \( t' \) earlier than \( t \) such that \( \overline{W}(t') > 0 \) and \( \sum_{t=1}^{v_i} y_i(t) > 0 \) when AllocateRLM(\( t \)) is in its loop iteration for \( t \) and is also in the loop of Routine(\( t \)).

In Allocate-A(i), we can conclude that \( \overline{W}(t') > 0 \) upon completion of Fully-Utilize(\( i \)) since \( t' \geq u_i \geq v_i \) and by Lemma A.2. We also have that \( \sum_{t=1}^{v_i} y_i(t) > 0 \) upon completion of Fully-Utilize(\( i \)) by Lemma A.3(3) and Lemma A.3(2) holds. By Lemma A.3(3) and Lemma A.3, we also have \( \overline{W}(t) = 0 \) currently from \( \sum_{t=1}^{v_i} y_i(t) > 0 \). In Allocate-B(i), one more function Fully-Utilize(i) is called and if such \( t' \) exists when Routine(\( t \)) is called in AllocateRLM(\( t \)), we have \( t \leq \omega \) when \( \Delta > 0 \) and AllocateR-LM(\( t \)) takes an effect on \( y_i(t) \) by Lemma A.4. We can come to the same conclusion in Allocate-B(\( i \)) that \( y_i(t') = k_i \) and \( \overline{W}(t) = 0 \) using an additional Lemma 3.3.

Finally, in both Allocate-A(\( i \)) and Allocate-B(\( i \)) we have the same observation as we have made in Fully-Utilize(\( i \)): \( W(t) - y_i(t) > W(t') - y_i(t') \) and there must exist such a task \( T_v \) that \( y_v(t') < y_v(t) \); otherwise, we can not have that inequality. In the subsequent loop iterations of Routine(\( t \)), \( \overline{W}(t) \) becomes greater than 0 but \( \overline{W}(t) < \Delta \leq k_i - y_i(t) \). We still have \( W(t) - y_i(t) = C - \overline{W}(t) - y_i(t) > W(t') - k_i = W(t') - y_i(t') \). Such \( T_v \) can still be found.

4) **Allocate-X(\( i \)):** Allocate-B(\( i \)) and Allocate-A(\( i \)) are respectively presented as Algorithm 2 and Algorithm 8. They will be respectively called by the optimal scheduling algorithm LDF(S) in Section III and the greedy algorithm GreedyRLM in Section IV. Let \( A \) denote the set of the tasks that have been fully allocated so far excluding \( T_i \).

**Lemma A.5:** Upon every completion of the allocation algorithm Allocate-A(\( i \)) or Allocate-B(\( i \)), the workload \( W(t) \) at every time slot is not decreased in contrast to the one just before the execution of this allocation algorithm. In other words, if \( W(\overline{t}) > 0 \) upon its completion, \( W(t) > 0 \) just before its execution.

**Proof:** We observe the resource allocation state on the whole. Fully-Utilize(\( i \)) never change the allocation of any \( T_j \) in \( \mathcal{A} \) at every time slot. To further prove Lemma A.5, we only need to show that, in the subsequent execution of Allocate-A(\( i \)) or Allocate-B(\( i \)), the total allocation at every time slot is no less than the total allocation of \( \mathcal{A} \) at \( t \) upon completion of Fully-Utilize(\( i \)), i.e., \( \sum_{t \in \mathcal{A}} y_i(t) \).

In Allocate-A(\( i \)), when AllocateRLM(\( i \), \( \eta_1 \)) is dealing with a time slot \( t \), it does not decrease the allocation at every time slot in \([v_{d_i} + 1, d_i] \) by Lemma A.2, where \( v_i \leq v_{d_i} \). The only operations of decreasing the workload at \( t \) in \([1, v_{d_i}] \) form lines 6-10 of AllocateRLM(\( t \)) and they only decrease and change the allocation of \( T_i \). The allocation of \( T_j \in \mathcal{A} \) at every time slot in \([1, v_{d_i}] \) is still the one upon completion of Fully-Utilize(\( i \)) since we always have \( t' \geq u_i \geq v_i \) in
Route() by Lemma A.6(1). Hence, the final workload of $A$ at $t \in [1, v_d]$ is at least the same as the one upon completion of Fully-Utilize(). The lemma holds in Allocate-A(i).

In Allocate-B(i), we need to additionally consider a call to Fully-Allocate(i). By Lemma A.3 the lemma holds upon completion of Fully-Allocate(i). Since $y_i(t) = k_i$ for all $t \in [\omega + 1, d_i]$ by Lemma A.6 we have that the subsequent call to AllocateRLM() will take no effect on the workload at $t \in [\omega + 1, d_i]$ and the lemma holds in $[\omega + 1, d_i]$ upon completion of Allocate-A(i). Since $T_i$ is allocated some more $\overline{W}(t)$ machines at $\omega$ in the loop iteration of Fully-Allocate(i) for $\omega$, $\overline{W}(t)$ becomes 0 and will also be 0 upon completion of AllocateRLM() as we described in its executing process. Upon completion of Fully-Allocate(i), the allocation of $T_i$ in $[1, \omega - 1]$ is still the one done by Fully-Utilize(i) by the definition of $\omega$ and the total allocation of $A$ at every time slot in $[1, \omega - 1]$ is no less than the one upon completion of Fully-Utilize(i). Then, AllocateRLM() will change the allocation in $[1, \omega - 1]$ in the same way as it does in Allocate-A(i) and the lemma holds in $[1, \omega - 1]$. Hence, the lemma holds in Allocate-B(i).

Lemma A.6: Upon completion of Allocate-A(i) or Allocate-B(i), if there exists a time slot $t''$ that satisfies: $t'' \in (t_{m-1}', t])$ (here assume that $T_i \in A_m$) such that $\overline{W}(t'') > 0$ in Allocate-A(i), or $t'' \in [1, t]$ such that $\overline{W}(t'') > 0$ in Allocate-B(i), we have that

1. in the case that $\text{len}_i < d_i - t'' + 1$, $\sum_{i=1}^{d_i-1} y_i(T) = 0$;
2. in the case that $\text{len}_i \geq d_i - t'' + 1$, $T_i$ is allocated $k_i$ machines at each time slot $t \in [t''', d_i]$;
3. the total allocation $\sum_{i=t''}^{d_i} y_j(T)$ for every $T_j \in A$ in $[t'', d_i]$ is still the same as the one just before the execution of Allocate-A(i) or Allocate-B(i).

Proof: We observe the stop state of AllocateRLM(). In the case that $\text{len}_i < d_i - t'' + 1$, if $\sum_{i=1}^{d_i-1} y_i(T) > 0$, there exists a time slot $t$ such that $y_i(t) < k_i$. The loop of Routine($\Delta$, $\eta_1$, $\eta_2$) for $t$ would not stop with the current state by the condition given in Algorithm 4 (also described in Section III) and Section IV-A. Lemma A.6(1) holds. In the case that $\text{len}_i \geq d_i - t'' + 1$, we have that $\sum_{i=1}^{d_i-1} y_i(T) > 0$ and there is no time slot $t$ such that $y_i(t) < k_i$; otherwise, the loop of Routine($\Delta$, $\eta_1$, $\eta_2$) for $t$ would not stop with the current state. Lemma A.6(2) holds.

Now, we prove Lemma A.6(3). In Allocate-A(i), we discuss two cases on $t''$ just before the execution of the loop iteration of AllocateRLM($\Delta$, $\eta_1$) at every $t \in [t''', d_i]$: $\overline{W}(t'') = 0$ and $\overline{W}(t'') > 0$. We will prove $t'' \leq t'$ in the loop iteration of Routine() since the change of the allocation of $A$ only occurs between $t$ and $t'$ for the task $T_i$.

If there exists a certain loop iteration of AllocateRLM() at $t \in [t''', d_i]$ such that $\overline{W}(t'') = 0$ initially but $\overline{W}(t'') > 0$ upon completion of this loop iteration, this shows that there exists some operations that decrease the allocation at $t''$. Such operations only occur in lines 6-10 of AllocateRLM() and we have that $t'' \leq t_i$. Since $v_i < u_{i+1} \leq \cdots \leq u_d$, we have that $t'' \leq t'$ in the loop iteration of Routine() for every $t \in [t''', d_i]$ by Lemma A.3. Here, we also have that AllocateRLM() will do nothing in its loop iteration at $t''$ since $\sum_{i=1}^{d_i-1} y_i(T) = 0$ then. Hence, Lemma A.6(3) holds in this case. In the other case, $\overline{W}(t'') > 0$ just before the execution of every loop iteration of AllocateRLM() for every $t \in [t''', d_i]$, and we have that $t'' < t'$. Just before the execution of AllocateRLM(), we also have that either $y_i(t'') = k_i$ or $\sum_{i=1}^{d_i-1} y_i(T) = 0$ by Lemma A.3. Then, the loop iteration of AllocateRLM() will take no effect on the allocation at $t''$. Hence, Lemma A.6(3) also holds in this case.

In Allocate-B(i), the additional function Fully-Allocate() will be called and we will discuss the positions of $t''$ in $[\omega + 1, d_i]$. If $t'' \in [\omega + 1, d_i]$, AllocateRLM() will take no effect on the allocation at every time slot in $[t'', d_i]$ and we have that $\overline{W}(t'') > 0$ upon completion of Fully-Allocate(i). By Lemma A.3 and Lemma A.3 we have either $y_i(t'') = k_i$ or $\sum_{i=1}^{d_i-1} y_i(T) = 0$ upon completion of Fully-Utilize(i). In the latter case, the call to Fully-Allocate(i) will cannot take any effect on the allocation at every time slot in $[1, d_i]$ since $\Omega = 0$ there. In the former case, we always have $t'' \geq t'$ in the loop iteration of Fully-Allocate(i) for every $t \in [t''', d_i]$ and Fully-Allocate(i) cannot change the allocation of any task at its loop iteration for $t''$. Hence, Lemma A.6(3) holds when $t'' \in [\omega + 1, d_i]$. If $t'' = \omega$, AllocateRLM() does not decrease the allocation at $\omega$. We also have that $\overline{W}(t'') > 0$ upon completion of Fully-Allocate(i), and then either $y_i(t'') = k_i$ or $\sum_{i=1}^{d_i-1} y_i(T) = 0$. AllocateRLM() will also do nothing at $t''$ (i.e., $\omega$) and Lemma A.6(3) holds when $t'' = \omega$.

If $t'' \in [1, \omega - 1]$, we first observe the effect of Fully-Allocate(i). In the case that $\overline{W}(t'') = 0$ upon completion of Fully-Allocate(i) but $\overline{W}(t'') > 0$ upon completion of AllocateRLM(), we have that the allocation in $[1, t'']$ have ever been decreased and there exists a time slot $t'$ such that $\overline{W}(t') > 0$ when AllocateRLM() is in its loop iteration at a certain $t \in [1, \omega]$, where $t' < u_t \leq t < \omega$. By Lemma A.2 this $u_t$ is also not fully utilized upon completion of Fully-Allocate(i). Further, by Lemma A.3 upon completion of every loop iteration of Fully-Allocate(i) at $t \in [1, d_i]$, $u_t$ is not fully utilized and Lemma A.6(3) holds in this case. In the other case, $\overline{W}(t'') > 0$ upon completion of Fully-Allocate(i). By Lemma A.2 $t''$ is not fully utilized upon every completion of its loop iteration at $t \in [\omega, d_i]$, $\overline{W}(t'') > 0$ and the exchange of allocation of $A$ only occurs in two time slots $t$ and $t'$ in Routine() that are in $[t'', d_i]$. Hence, Lemma A.6(3) holds in this case. Further, Lemma A.6(3) holds upon completion of Fully-Allocate(i). As we analyze in Allocate-A(i), Lemma A.6(3) still holds upon completion of AllocateRLM().

B. Optimal Scheduling Algorithm

This section is supplementary to Section III.

Proof: (of Lemma 5.1) By induction. When $m = 0$, the lemma holds trivially. Assume that this lemma holds when $m = l$. If $\lambda_{l+1}(S) - \lambda^*_l(S) < C(t_{L-l-1} - t_{L-l-1})$, it means that with the capacity constraint in $[t_{L-l-1} + 1, d_i]$, $S$ can still
utilize at most $\lambda_i(S)$ resources in $[\tau_{L-i} + 1, d]$ and this lemma holds; otherwise, after $S$ has utilized the maximal amount of $\lambda^*_C(S)$ resources in $[\tau_{L-i} + 1, d]$, it can only utilize at most $C(\tau_{L-i} - \tau_{L-i-1})$ resources in $[\tau_{L-i-1} + 1, \tau_{L-i}]$. The lemma holds when $m = 1 + 1$.

Proof: (of Lemma 3.2) By the implication of the parameter $\lambda^*_C(S)$ in Lemma 3.1 after $S$ has optimally utilized the machines in $[\tau_m + 1, d]$, if there exists a feasible schedule for $S$, the total amount of the remaining demands in $S$ should be no less than the capacity $C_{\tau_m}$ in $[1, \tau_m]$.

In the following, we let $A$ always denote the set of tasks that have been fully allocated so far excluding $T_i$ and we will prove Proposition 3.1. We first give the following lemma:

Lemma A.7: Let $m \in [L]^+$. Suppose $T_i \in \mathcal{S}_{L-m+1}$ is about to be allocated. If we have the relation that $\overline{W}(1) \geq \overline{W}(2) \geq \cdots \geq \overline{W}(\tau_{L-m+1})$ before the execution of Allocate-B(i), such relation on the available machines at each time slot still holds after the completion of Allocate-B(i).

Proof: We observe the executing process of Allocate-B(i). Allocate-B(i) will call three functions Fully-Utilize(i), Fully-Allocate(i) and AllocateRLM(i, 1). In every call to those functions, the time slots $t$ will be considered from the deadline of $T_i$ towards earlier time slots. During the execution of Fully-Utilize(i), the allocation to $T_i$ at $t$ is $y_i(t) = \min\{k_i, D_t - \sum_{t'=1}^{t-1} y_i(t'), \overline{W}(t)\}$. Before time slots $t$ and $t + 1$ are considered, we have $\overline{W}(t) \geq \overline{W}(t + 1)$. Then, after those two time slots are considered, we still have $\overline{W}(t) \geq \overline{W}(t + 1)$.

During the execution of Allocate-B(i), let $t'$ always denote the current time slot such that $\overline{W}(t') > 0$ and $\overline{W}(t' + 1) = 0$ if such time slot exists. $t'$ is also unique when the relation on the available machines holds. By Lemma 3.3 if $\Omega > 0$ at the very beginning of the execution of Fully-Allocate(i), we have $y_i(1) = \cdots = y_i(t') = k_i$ upon completion of Fully-Utilize(i) and the allocation of $T_i$ at every time slot in $[1, t']$ will not be changed by Fully-Allocate(i) since $\Delta = 0$ then. When Fully-Allocate(i) is considering a time slot $t (t > t')$, it will transfer partial allocation of $T_i$ at $t$ to the time slot $t'$. If $t'$ becomes fully utilized, $t' - 1$ becomes the current $t'$ and Routine(i) will make time slots fully utilized one by one from $t'$ towards earlier time slots. In addition, the time slot $t$ will again become fully utilized by line 5 of Fully-Allocate(i), i.e., $\overline{W}(t) = 0$. The allocation at every time slot in $[1, t' - 1]$ are still the one upon completion of Fully-Utilize(i) and the allocation at $t'$ is never decreased. Hence, the relation on the available machines still holds upon completion of every loop iteration of Fully-Allocate(i) for $t \in [1, d_i]$.

From the above, we have the following facts upon completion of Fully-Allocate(i): (1) the allocation of $T_i$ in $[1, t']$ is still the one upon completion of Fully-Utilize(i); (2) the allocation of $A$ in $[1, t' - 1]$ is still the one just before the execution of Allocate-B(i), and (3) the allocation of $A$ at $t'$ is not decreased in contrast to the one just before the execution of Allocate-B(i). Hence, we have that $C - \sum_{T_i \in A} y_i(1) \geq C - \sum_{T_i \in A} y_i(2) \geq \cdots \geq C - \sum_{T_i \in A} y_i(t')$.

Upon completion of every loop iteration of AllocateRLM(i) at $t \in [t' + 1, d_i]$, $u_t = t'$ or $t' + 1$. When AllocateRLM(i) is considering $t$, the time slots from $t'$ towards earlier time slots will become fully utilized in the same way as Fully-Allocate(i). Hence, the relation on the number of available machines holds obviously in $[t' + 1, d_i]$ since every time slot there is fully utilized. Since $v_i < u_t$ by Lemma A.3 the allocation of $A$ in $[1, v_i]$ has not been changed since the execution of AllocateRLM(i), and we still have that the relation on the number of available machines holds in $[1, v_i]$ in both the case where $v_i < t'$ and the case where $v_i = t'$. Here, if $v_i < t'$, the allocation at $t'$ is never decreased by AllocateRLM(i) and further if $t' - 1 \geq v_i + 1$, the allocation at every $t \in [v_i + 1, t' - 1]$ has also not been changed so far by AllocateRLM(i). Hence, we can conclude that Lemma A.7 holds upon completion of the loop iteration at $t' + 1$. Then, if $t' = v_{i+1}$, AllocateRLM(i) will not take an effect on the allocation in $[1, t']$ and the lemma holds naturally. Otherwise, $t' > v_{i+1}$ and the allocation of $T_i$ in $[v_{i+1} + 1, t']$ is still the one done by Fully-Utilize(i) and the allocation in $[v_{i+1} + 1, t']$ is also not decreased in contrast to the one upon completion of Fully-Utilize(i) (i.e., $\overline{W}(t) > 0$ then). By Lemma 3.3 we conclude that AllocateRLM(i) will not take an effect on the allocation in $[1, t']$ and the lemma holds upon completion of AllocateRLM(i).

Lemma A.8: Upon completion of Allocate-B(i) for a task $T_i \in \mathcal{S}_{L-m+1}$ ($m \in [L]^+$), we have that

1. Let $t_1, t_2 \in [1, \tau_{L-m+1}]$ and $t_1 < t_2$; then, $\overline{W}(t_1) \geq \overline{W}(t_2)$;
2. $T_i$ is fully allocated;
3. $\lambda^*_C(A \cup \{T_i\})$ resources in $[\tau_{L-j} + 1, \tau_L]$ have been allocated to $A \cup \{T_i\}$ ($1 \leq j \leq L$).

Proof: By induction. Initially, $A = \emptyset$. When the first task $T_i = T_{L-1}$ in $\mathcal{S}_L$ is being allocated, Lemma A.3(1) holds by Lemma A.7. Since the lines 1-3 of Algorithm 11 will allocate $\min\{k_i, D_t - \sum_{t'=1}^{d_i} y_i(t'), \overline{W}(t)\}$ machines to $T_i$ from its deadline towards earlier time slots, and the single task can be fully allocated definitely, the lemma holds. We assume that when the first $l$ tasks in $\mathcal{S}_L$ have been fully allocated, this lemma holds.

Assume that this lemma holds just before the execution of Allocate-B(i) for a task $T_i \in \mathcal{S}_{L-m+1}$. We now show that this lemma also holds upon completion of Allocate-B(i). By Lemma A.7 Lemma A.3(1) holds upon completion of Allocate-B(i). Allocate-B(i) makes no change to the allocation of $A$ in $[\tau_{L-m+1} + 1, \tau_L]$ due to the deadline $d_i$ and Lemma A.3(3) holds in the case that $j \in [m-1]^+$ by the assumption. Here, if $m = 1$, the conclusion above holds trivially. Let $t'$ always denote the current time slot such that $\overline{W}(t') > 0$ and $\overline{W}(t' + 1) = 0$ if such time slot exists. If such time slot does not exist upon completion of Allocate-B(i), $T_i$ has been fully allocated since $S$ satisfies the boundary condition. Now, we discuss the case that $\overline{W}(1) > 0$ upon completion of Allocate-B(i). By Lemma A.6 we know that $T_i$ has also been fully allocated, and Lemma A.8(2) holds upon completion of Allocate-B(i). Assume that $t' \in (\tau_{L-t' - 1} + 1, \tau_{L-t' + 1}]$. 
By the definition of $t'$, Lemma A.8(3) holds in the case that $m \leq j \leq l' - 1$ obviously. By Lemma A.6, $T_i$ has already optimally utilized the resource in $[\tau_{L-1} \tau_L]$ for all $l' \leq j \leq \tau_L$ and so has the set $A$ together with the assumption. Lemma A.8(3) holds.

Proof: (of Proposition 4.1) By Lemma A.8(2), the proposition holds when all the tasks in $S$ have been considered in the algorithm LDF(S).

Lemma A.9: The time complexities of GreedyRLM and LDF(S) are $O(n^2)$.

Proof: The time complexity of Allocate-A(i) comes from AllocateRLM($i$, $\eta_i$) and the time complexity of Allocate-B(i) comes from Fully-Allocate(i) or Allocate-RLM(i). In Allocate-B(i), in the worst case, Fully-Allocate(i) and AllocateRLM(i) have the same time complexity from the execution of Routine() for every time slot $t \in [1, d_i]$. In the call to AllocateRLM() for every task $T_i \in \mathcal{A}$, the loop iteration there for all $t \in [1, d_i]$ needs to seek for the time slot $t'$ and the task $T_{i'}$ at most $d_i$ times. The time complexity of seeking for $t'$ is $O(d_i)$; the time complexity of seeking for $T_{i'}$ is $O(n)$. Since $|\mathcal{T}| = n$, $d_i \leq d$, and $D_i \leq D$, we have that the time complexities of those algorithms are $O(n d D n)$.

Given the assumption that the maximal deadline $d$ and the maximal parallelism bound $k$ of tasks is finitely bounded, the time complexities of those algorithms are $O(n^2)$ since $D \leq k d$.

C. Upper Bound of Greedy Algorithm

Proof: (of Proposition 4.1) Let us consider a special instance:

1. Let $D = \{d_1', d_2'\}$ be the set of deadlines, where $d_2', d_1' \in \mathbb{Z}^+$ and $d_2' > d_1'$. Let $D_1 = \{T_i \in \mathcal{T} | d_j = d_i' \} (1 \leq i \leq 2)$.

2. Let $\epsilon \in (0, 1)$ be small enough. For all $T_i \in D_1$, (a) $v_i' = 1 + \epsilon$ and $k_i = 1$; (b) There are $C \cdot d_i'$ tasks $T_i$ with $D_i = 1$.

3. $v_i' = 1$, $k_i = 1$ and $D_i = d_2' - d_1' + 1$ for all $T_i \in D_2$.

GREEDY will always fully allocate resource to the tasks in $D_1$, with all the tasks in $D_2$ rejected to be allocated any resource. The performance guarantee of GREEDY will be no more than $\frac{C \cdot d_i'}{\frac{\epsilon}{1+\epsilon}(d_1'-1)+1}\cdot(\frac{d_2'-d_1'+1}{2})$. Further, with $\epsilon \to 0$, this performance guarantee approaches $\frac{d_i'}{d_2'}$. In this instance, $s = \frac{d_2'}{d_2'-d_1'+1}$ and $s^{-1} = \frac{d_1'-1}{d_2'}$. When $d_2' \to +\infty$, $\frac{d_i'}{d_2'} \to s^{-1}$. Hence, this proposition holds.

D. Theorem: A Novel Analysis Technique

In this section, we prove Theorem 4.1

Definition. We first give a sufficient condition for a time interval $[d', d]$ to be optimally utilized by $\mathcal{T}$, i.e., to be such that the maximal amount of the total demand of $\mathcal{T}$ that could be executed over $[d', d]$ is executed:

Definition 1: The interval $[d', d]$ is optimally utilized by $\mathcal{T}$ if, for all tasks $T_i \in \mathcal{T}$ with $d_i \geq d'$, one of the following two conditions is satisfied:

1. if $\sum_{t=d}^{T} y_i(t) = D_i$.
2. if $\sum_{t=d}^{T} y_i(t) = D_i$.

In particular, if there exists no task $T_i \in \mathcal{T}$ such that $d_i \geq d'$, $[d', d]$ is optimally utilized by $\mathcal{T}$ trivially.

Scheduling tasks with relaxed constraints. Our analysis is retrospective. We will in this section treat $T_{m+1}$ as a real task, and consider the welfare maximization problem through scheduling the following tasks:

1. $T_{m+1} = \{T_{m+1} \mid T_i \in \mathcal{U}_{j=m+1} \mathcal{A}_j\}$
2. $\mathcal{F}_{m+1} = \{T_{m+1} \mid T_i \in \mathcal{U}_{j=m+1} \mathcal{A}_j\}$
3. $\mathcal{N} = \mathcal{U}_{j=m+1} \mathcal{R}_j$

Here, $m \in [K]$ and we relax several restrictions on the tasks. Specifically, partial execution can yield linearly proportional value, that is, if $T_i$ is allocated $\sum_{t=d_i}^{T} y_i(t) < D_i$ resources by its deadline, a value $\frac{\sum_{t=d_i}^{T} y_i(t)}{D_i}$ will be added to the social welfare. The parallelism bound $k_i$ of the tasks $T_i \in \mathcal{F}_{m+1}$ is reset to $C$.

Let $\mathcal{T}_{m+1} = \{T_{m+1} \mathcal{T}_{m+1} \mathcal{U} \mathcal{F}_{m+1} \mathcal{U} \mathcal{N}\}$. Denote by $OPT_T^{[1, t_{m+1}]}$ and $OPT_T^{[1, t_{m+1}]}$ the optimal social welfare by scheduling $\mathcal{T}_{m+1}$ in the segmented timescales $[1, t_{m+1}]$ and $[t_{m+1}, 1]$. The connection of the problem above with our original problem is that with the relaxation of some constraints of tasks $OPT_T^{[1, t_{m+1}]}$ is an upper bound of the optimal social welfare in our original problem. In the following, we will bound $OPT_T^{[1, t_{m+1}]}$ by bounding $OPT_T^{[1, t_{m+1}]}$ and $OPT_T^{[1, t_{m+1}]}$ in $OPT_T^{[1, t_{m+1}]}$ in order to bound $OPT_T^{[1, t_{m+1}]}$. We therefore assume that $t_{m+1} > t_{m+1}^h$ subsequently. The lemma below shows that the bound on $OPT_T^{[1, t_{m+1}]}$ can be obtained through bounding each $OPT_T^{[1, t_{m+1}]}$ in $OPT_T^{[1, t_{m+1}]}$.

Lemma A.10: $OPT_T^{[1, t_{m+1}]} \leq OPT_T^{[1, t_{m+1}]} + OPT_T^{[1, t_{m+1}]} (m \in [K])$

Proof: Consider an optimal schedule achieving $OPT_T^{[0, t_{m+1}]}$. If a task $T_{K+1, t_{m+1}} \in \mathcal{T}_{m+1} \mathcal{U} \mathcal{F}_{m+1}$ is allocated more than $D_{K+1, t_{m+1}}$ resources in the time slot interval $[1, t_{m+1}^h]$, we transfer the part larger than $D_{K+1, t_{m+1}}$ (at most $D_{K+1, t_{m+1}}^h$) to $[t_{m+1}^h, 1]$ and in the meantime transfer to $[1, t_{m+1}^h]$ partial allocation of $\mathcal{N}$ in $[t_{m+1}^h, 1]$ if the allocation transfer in the former needs to occupy the allocation of $\mathcal{N}$. The former is feasible in that the total allocation of $T_{K+1, t_{m+1}}$ in $[t_{m+1}^h, 1]$ can be up to $D_{K+1, t_{m+1}}^h$ without violating the constraints of the deadline and parallelism bound of $T_{K+1, t_{m+1}}$. The latter is feasible since the parallelism bound there is $C$. Then, this optimal schedule can be transferred into a feasible schedule of $\mathcal{T}_{m+1}$ in $[1, t_{m+1}]$ plus a feasible schedule of $\mathcal{T}_{m+1}$ in $[t_{m+1}, 1]$. The lemma holds.

Bound of time slot intervals. In this section, we consider the following schedule of $\mathcal{T}_{m+1}$ ($m \in [K]$). Whenever $\mathcal{T}_{m+1}$ is
concerned, the allocation to the tasks of \( T_{m+1} \) at every time slot \( t \in [1, t_{m+1}] \) is \( T_{K_{m+1},i}^{[t,t]} = y_i(t) \), as is done by Greedy in \([1, t_{m+1}] \) for the set of tasks \( T \). Note that the tasks in \( N \) are all rejected. We will study how to bound \( OPT^{[t_{m+1}, t+1}_{m+1}, t_{m+1}] \) using the above allocation of \( T_{m+1} \).

We first observe, in the next two lemmas, that schedule can achieve \( OPT^{[t_{m+1}, t+1}_{m+1}, t_{m+1}] \).

**Lemma A.11:** \[ OPT^{[t_{m+1}, t+1}_{m+1}, t_{m+1}] = \sum_{T_i \in \cup_{j=1}^{K} A_j} v(T_{m+1}) . \]

**Proof:** When \( m = K \), \( \delta_i \leq t_{m+1}^{[t_{m+1}, t+1]} \) for all \( T_i \in N \) and \( F_{m+1} = \emptyset \). The allocation of \( N \) in \([t_{m+1}, t+1] \) yields no value. By Feature 4.2, the lemma holds.

For ease of exposition, let \( \mathcal{F}_{m+1} = \{T_{K_{m+1},i}^{[t_{m+1}, t+1]} : T_i \in \mathcal{A}_{m+1}\} \) and \( \mathcal{F}_{m+1}^- = \{T_{K_{m+1},i}^{[t_{m+1}, t+1]} : T_i \in \cup_{j=1}^{K} A_j\} \). We also let \( N_{m}^+ = \cup_{j=1}^{m-1} \mathcal{R}_j \), and \( N_{m}^+ = \cup_{j=1}^{m-1} \mathcal{R}_j \). Here, \( \mathcal{F}_{m+1} = \mathcal{F}_{m+1}^+ \cup \mathcal{F}_{m+1}^- \) and \( N = N_{m} \cup N_{m}^- \).

**Lemma A.12:** With the relaxed constraints of tasks, we have for all \( m \in [K-1] \) that \( OPT^{[t_{m+1}, t+1]}_{m+1} \) can be achieved by the following schedule:

1. \( D_{K_{m+1},i}^{[t_{m+1}, t+1]} \) resources are allocated to every task \( T_{K_{m+1},i}^{[t_{m+1}, t+1]} \) in \( \mathcal{F}_{m+1}^- \);
2. For the unused resources in \([t_{m+1}, t+1] \), we execute the following loop until there is no available resources or \( \mathcal{F}_{m+1}^+ \cup N_{m}^+ \) is empty: select a task in \( \mathcal{F}_{m+1}^+ \cup N_{m}^+ \) with the maximal marginal value and allocate as many resources as possible with the constraint of deadline; delete this task from \( \mathcal{F}_{m+1}^+ \cup N_{m}^+ \).

**Proof:** With the constraint of deadlines, the allocation of \( N_{m}^+ \) in \([t_{m+1}, t+1] \) yields no value due to \( \delta_i \leq t_{m+1} \). As a result of the order that Greedy considers tasks, the tasks of \( T_{m+1} \) will have higher marginal values than the tasks in \( N_{m}^+ \cup \mathcal{F}_{m+1}^- \). By Feature 4.2, the lemma holds.

We now bound \( OPT_{m+1}^{[t_{m+1}, t+1]} \) \((m \in [K-1])\) using the allocation of \( T_{m+1} \) specified above. With abuse of notation, for a set of accepted tasks \( \mathcal{A} \), let \( A \) denote both the area occupied by the allocation of the tasks of \( \mathcal{A} \) in some slot interval and the size of this area.

**Lemma A.13:** There exist \( K \) areas \( C_1, C_2, \ldots, C_K \) from \( T_{K-1}^{[t_{m+1}, t+1]} \cup F_{K-1}^{+} \) such that

1. the size of each \( C_{m+1} \) is \( r \cdot C \cdot (t_{m+1} - t_{m+1}^{[t_{m+1}, t+1]}) \) \((m \in [K-1])\);
2. every \( C_{m+1} \) is obtained from the area \( T_{m+1}^{[t_{m+1}, t+1]} \cup F_{m+1} = \sum_{j=0}^{m} C_j \), where \( C_0 = 0 \), and is the part of that area with the maximal marginal value.

**Proof:** When \( m = 0 \), by Feature 4.1, \( \sum_{T_i \in A_j} D_{K_{m+1},i}^{[t_{m+1}, t+1]} \geq r \cdot C \cdot t_{m+1}^{[t_{m+1}, t+1]} \) and the lemma holds. Assume that when \( m \leq l \) \((l \geq 0)\), the lemma proves. Now, we prove the lemma holds when \( m = l + 1 \). By Feature 4.1 \( T_{l+1}^{[t_{m+1}, t+1]} \cup F_{l+2}^{+} \geq r \cdot C \cdot t_{l+1}^{[t_{m+1}, t+1]} \).

By the assumption, \( \sum_{j=0}^{l+1} C_j = r \cdot C \cdot t_{l+1}^{[t_{m+1}, t+1]} \) and \( \cup_{j=0}^{l+1} C_j \subseteq T_{l+1}^{[t_{m+1}, t+1]} \cup F_{l+1}^{+} \), the lemma holds for \( m = l + 1 \).}

We emphasize here that the tasks in \( A_1 \cup \mathcal{R}_1 \cup A_2 \cup \cdots \cup \mathcal{R}_K \) have been considered and sequenced in the non-increasing order of the marginal value. Recall the composition of \( T_{m+1}^{[t_{m+1}, t+1]} \) and the allocation to \( T_{m+1}^{[t_{m+1}, t+1]} \cup F_{m}^{+} \) in fact corresponds to the allocation of \( T_{m+1}^{[t_{m+1}, t+1]} \) in \([1, t_{m+1}^{[t_{m+1}, t+1]}] \). Let \( V_{m+1} \) be the total value associated with the area \( C_{m+1} \). Since \( T_{m+1}^{[t_{m+1}, t+1]} \cup F_{m}^{+} - \sum_{j=0}^{m} C_j \geq 0 \) and \( T_{m+1}^{[t_{m+1}, t+1]} \cup F_{m}^{+} \subseteq \cup_{T_i \in \mathcal{A}_{m+1}} \) by the way that we obtain \( C_{m+1} \) in Lemma A.13 and the optimal schedule that achieves \( OPT^{[t_{m+1}, t+1]}_{m+1} \) in Lemma A.12 we have that

\[
\frac{V_{m+1}'}{r \cdot C \left( D_{m+1}^{[t_{m+1}, t+1]} \right)^m} \geq OPT^{[t_{m+1}, t+1]}_{m+1} \mid C_{m+1} \mid .
\]

The conclusion above in fact shows that the average marginal value of \( C_{m+1} \) is no less than the one in an optimal schedule, which uses the same principle as the greedy algorithm in the knapsack problem in Section 2. Finally, we have that \( OPT^{[t_{m+1}, t+1]}_{m+1} \leq \frac{V_{m+1}'}{r} \) for all \( m \in [K-1] \).

By Lemma A.10, \( OPT^{[t_{m+1}, t+1]}_{m} \leq \sum_{j=1}^{K} V_{j}' / r \). By Lemma A.11 we further have that

\[
OPT^{[t_{m+1}, t+1]}_{m} \leq \sum_{j=1}^{K} V_{j}' / r + OPT^{[t_{m+1}, t+1]}_{m+1} \]

Hence, Theorem 4.1 holds.

**E. GreedyRLM**

GreedyRLM is presented as Algorithm [7]. We will prove that Feature 4.1 and Feature 4.2 hold in GreedyRLM, where \( r = \frac{1}{s} \). Then, we have that GreedyRLM gives an \( \frac{1}{s} \) approximation to the optimal social welfare.

**Proposition A.1:** Upon completion of GreedyRLM, Feature 4.1 holds in which \( r = \frac{1}{s} \).

**Proof:** Upon completion of the \( m \)-th phase of GreedyRLM, consider a task \( T_i \in \cup_{j=1}^{m} \mathcal{R}_j \) such that \( d_i = c_m \). Since \( T_i \) is not accepted when being considered, it means that \( D_i \leq \sum_{i \leq d_i, \min \{ k_i, \bar{W} (t) \} \text{ at that time } \text{ and } \text{there are at most } \bar{W} (t) \geq k_i \text{ in } [1, c_m] \}. \) We assume that the number of the current time slots \( t \) with \( \bar{W} (t) \geq k_i \) is \( \mu \). Since \( T_i \) cannot be fully allocated, we have the current resource utilization in \([1, c_m] \) is at least

\[
\frac{C \cdot d_i - \mu C - (D_i - \mu k_i)}{C \cdot d_i} \geq \frac{C \cdot d_i - D_i - (\bar{W} - 1)(C - k_i)}{C \cdot d_i} \geq \frac{C (d_i - k_i) + (C - k_i) + (\bar{W} - 1)(C - k_i)}{C \cdot d_i} \geq \frac{s - 1}{s} \geq r.
\]
We assume that \( T_{i} \in \mathcal{R}_{h} \) for some \( h \in [m]^{+} \). Allocate-A\((j)\) consists of two functions: Fully-Utilize\((j)\) and AllocateR-RLM\((j, 0)\). Fully-Utilize\((j)\) will not change the allocation to the previous accepted tasks at every time slot. In AllocateR-RLM\((j, 0)\), the operations of changing the allocation to other tasks happen in its call to Routine\((\Delta, 0, 1)\). Due to the function of lines 9-11 of Routine\((\Delta, 0, 1)\), after the completion of the \( h \)-th phase of GreedyRLM, the subsequent call to Allocate-A\((j)\) will never change the current allocation of \( \cup_{j=1}^{T} A_{j} \) in \([1, c_{m}]\). Hence, if \( t_{m}^{h} = c_{m} \), the lemma holds with regard to \( \cup_{j=1}^{m} A_{j} \) \((m \geq h)\); if \( t_{m}^{h} > c_{m} \), since each time slot in \([c_{m} + 1, t_{m}^{h}]\) is fully utilized, the resource utilization in \([c_{m} + 1, t_{m}^{h}]\) is 1 and the final resource utilization will also be at least \( r \).

Lemma A.14: Due to the function of the threshold \( t_{m}^{h} \) and its definition, we have for all \( T_{i} \in \mathcal{A}_{m} \) that

1. \([t_{m}^{h} + 1, d] \) is optimally utilized by \( T_{i} \) upon completion of Allocate-A\((i)\), where \( m \leq j \leq K \);
2. \( D_{K+1,m}^{[t_{m}^{h}+1, t_{m}^{h}+1]}(i) \) is the definition of \( t_{j}^{h} \). By Lemma A.9 and the definition of \( t_{j}^{h} \). Lemma A.5, the time slots \( t_{m}^{h} + 1, \ldots, t_{K}^{h} + 1 \) are not fully utilized upon completion of Allocate-A\((i)\) by Lemma A.5.

Proof: The time slots \( t_{m}^{h} + 1, \ldots, t_{K}^{h} + 1 \) are not fully utilized upon completion of Allocate-A\((i)\). The total allocation of \( T_{i} \) in \([t_{m}^{h} + 1, d] \) keeps invariant by Lemma A.6.\((3)\) for \( h \leq j \leq K \). Due to the function of the threshold parameter \( t_{m}^{h} \) in the \( h \)-th phase of GreedyRLM (line 9 of Routine\((j)\)), when AllocateR-RLM\((i, 0)\) is dealing with a time slot \( t \in [t_{m}^{h} + 1, t_{m}^{h}] \), the allocation change of other tasks can only occur in \([t_{m}^{h} + 1, t] \). Hence, we have that the total allocation of \( T_{i} \) in \([t_{m}^{h} + 1, t] \) keeps invariant and the allocation of \( T_{i} \) at every time slot in \([1, t_{m}^{h}] \) keeps invariant. We can therefore conclude that the total allocation of \( T_{i} \) in each \([t_{m}^{h} + 1, t_{m}^{h}] \) keeps invariant upon completion of Allocate-A\((i)\). Hence the lemma holds.

Proposition A.2: \([t_{m}^{h} + 1, t_{m}^{h+1}] \) is optimally utilized by \( \{T_{i}^{[0,t_{m}^{h}+1]} \mid T_{i} \in \mathcal{A}_{m} \} \).

Proof: We observe the allocation of a task \( T_{i} \in \mathcal{A}_{m} \) after the completion of GreedyRTL by Lemma A.2 and Lemma A.9. If \( t_{i} < d_{i} - t_{m}^{h} \), we have that \( D_{K+1,i}^{[0,t_{m}^{h}+1]} = 0 \).

If \( t_{i} - t_{m}^{h} < l_{i} \leq d_{i} - t_{m}^{h} \), we have that \( D_{K+1,i}^{[0,t_{m}^{h}+1]} = D_{i} - k_{i}(d_{i} - t_{m}^{h} + 1) \) and \( D_{K+1,i}^{[0,t_{m}^{h}+1]} = D_{K+1,i}^{[0,t_{m}^{h}+1]} \). If \( d_{i} - t_{m}^{h} < l_{i} \), we have that \( D_{K+1,i}^{[0,t_{m}^{h}+1]} = D_{K+1,i}^{[0,t_{m}^{h}+1]} = k_{i}(t_{m}^{h} + 1 - t_{m}^{h}) \). Hence the lemma holds by Definition A.1.

F. Dynamic Programming Algorithm

This section is for the DP algorithm in Section IV.

Proposition A.3: DP\((T)\) outputs a subset \( S \) of \( T = \{T_{1}, \ldots, T_{n}\} \) such that \( v(S) \) is the maximal value subject to the condition that \( S \) satisfies the boundary condition. The time complexity of DP\((T)\) is \( O(n^{d}C^{L}) \).

Proof: The proof is very similar to the one for knapsack problem in [21]. By induction, we need to prove that \( v(j) \) contains all the non-dominated pairs corresponding to feasible sets \( F \in \{T_{1}, \ldots, T_{n}\} \). When \( j = 1 \), the proposition holds obviously. Now suppose it hold for \( A(j - 1) \). Let \( F' \subseteq \{T_{1}, \ldots, T_{j} \} \) and \( F' \subseteq \{T_{i} \mid T_{i} \in \mathcal{F} \} \) satisfies the boundary condition. We claim that there is some pair \( (F, v(F)) \in (j) \) such that \( H(F) = H(F') \) and \( v(F) \geq v(F') \). First, suppose that \( T_{j} \notin F' \). Then, the claim follows by the induction hypothesis and by the fact that we initially set \( A(j) \) to \( A(j - 1) \) and removed dominated pairs. Now suppose that \( T_{j} \notin F' \). Then, the algorithm will add the pair \( (F_{1} \cup \{T_{j}\}, v(F_{1} \cup \{T_{j}\})) \) to \( A(j) \). Thus, there will be some pair \( (F', v(F')) \in (j) \) that dominates \( (F', v(F')) \). Since the size of the space of \( H(F) \) is no more than \( C^{L}T^{L} \), the time complexity of DP\((T)\) is \( nC^{L}T^{L} \).

Proof: (of Proof A.3) This proposition follows from Proposition A.3, Proposition 3.1 and Lemma A.9.

G. Algorithm for Machine Minimization

Proof: (of Proposition 4.4) If we have \( C = kn \) machines, there must exist a feasible schedule for \( T \) and \( kn \) is an upper bound of the minimal \( C \) so that the boundary condition can be satisfied. Then, the binary search can be used to find the minimal \( C \) so that the boundary condition is satisfied. The time complexity of this binary search is \( O(\ln kn) \). With Lemma A.9, the proposition holds.