Algorithms for Scheduling Deadline-Sensitive Malleable Tasks

Xiaohu Wu* and Patrick Loiseau*

***Abstract*— Due to the ubiquity of batch data processing in cloud computing, the fundamental problem of scheduling malleable batch tasks and its extensions have received significant attention recently. In this paper, we consider an important model in which 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 machines allocated to a task can vary over time without affecting its workload. For this model, we obtain two core results: a quantitative characterization of a sufficient and necessary condition such that a set of malleable batch tasks with deadlines can be scheduled on C machines, and a polynomial-time algorithm to produce such a feasible schedule. These core results provide a conceptual tool and an optimal scheduling algorithm that enable proposing new analyses and designs of algorithms and improving existing algorithms for extensive scheduling objectives.**

I. Introduction

Cloud computing has become the norm for a wide range of applications and, in the cloud, batch processing has become the most significant computing paradigm. With the ubiquity of batch data processing in cloud computing, many applications such as web search index update, monte carlo simulations or big-data analytics require the execution on computing clusters of batch jobs, i.e., non-real-time jobs with some flexibility on 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], [5], [6], [7], [8].

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 use the batch-processing paradigm to achieve a faster response [9], [10], [11]. Both on private clouds and on public clouds, organizations usually run business-critical tasks that are required to meet strict service-level agreements (SLAs) on latency, such as finishing before a certain deadline. Missing a deadline often has a significant consequences 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 metrics [9], [12], e.g., machine minimization; in the machine minimization objective, the algorithm will find the minimal amount of machines to complete a set of tasks by their respective deadlines. In fact, for better efficiency, companies such as IBM now integrate into their batch processing platform scheduling algorithms for various time metrics that are smarter than the popular dominant resource fairness strategy [13].

Hence, a timely and important algorithmic challenge posed here is to further our understanding of the problem of scheduling malleable batch tasks arising in cloud computing. In this paper, we reconsider the fundamental model of [3], [4] in which a set of n malleable batch tasks has to be scheduled 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. Beyond the analysis of this basic and important model, efforts have been devoted to its online version [5], [6], [7] and its extension [8], [9], [12] in which each task contains several subtasks with precedence constraints.

For the fundamental model in [3], [4] under the objective of maximizing social welfare (i.e., the sum of values of tasks completed before their deadline), Jain et al. have proposed an $(1 - \frac{C}{2C-k})(1-\epsilon)$-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 $\frac{C-k}{C} \cdot \frac{s-1}{s}$ in [4]. Here, k is the maximal parallelism bound of tasks, and s (≥ 1) is the slackness which intuitively 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 until its deadline to ensure full completion). In practice, the tasks tend to be recurring, e.g., they are scheduled periodically on an hourly basis [9]. Hence, we

*Xiaohu Wu and Patrick Loiseau are with the Department of Networking and Security, EURECOM, Sophia Antipolis, France Email: *firstname.lastname*@eurecom.fr

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 be viewed as a constant [14]. In this sense, the GreedyRTL algorithm has a polynomial time complexity of $\mathcal{O}(n^2)$.

Both algorithmic design techniques in [3], [4] are based on the theory of linear programming; they formulate the original problem as an integer program (IP) and relax the IP to 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 the LP to an integer solution of the 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 $\alpha \geq 1$) will be an upper bound of the optimal value of the IP, i.e., the social welfare of an optimal solution to the original problem. Therefore, the approximation ratio of the algorithm involved in the dual becomes clearly $1/\alpha$. 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 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 analyze it and 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, we will prove that answering the secondary question "*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?*" plays a **core** role in (i) understanding the above basic question, (ii) enabling the application of dynamic programming technique to the problem in [3], [4], and (iii) designing algorithms for other time metrics such as machine minimization. Intuitively, for any scheduling objective, an algorithm would be non-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 in the special case of scheduling on a single machine which shares some common features with the general case. In this case, the famous EDF (Earliest Deadline First) rule can lead to an optimal schedule [15]. 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 [15], [16], and (ii) as a significant principle in schedulability analysis for real-time systems [17].

Our contributions. In this paper, we 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. The 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 $\mathcal{S}$ with deadlines in terms of resource utilization. We first identify the basic constraints of malleable tasks and 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($\mathcal{S}$) that achieves such an optimal state. The LDF($\mathcal{S}$) algorithm has a polynomial time complexity of $\mathcal{O}(n^2)$ and is different from the EDF algorithm that gives an optimal schedule in the single-machine case.

Applications. The above core results have applications in several new or existing algorithmic design and analysis problems for scheduling malleable tasks under extensive objectives. In particular, we provide:

(i) an improved greedy algorithm GreedyRLM with an approximation ratio $\frac{s-1}{s}$ 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 $\mathcal{O}(\max\{n^2, nC^LM^L\})$;

(iii) the first exact algorithm for the machine minimization problem with a polynomial time complexity of $\mathcal{O}(n^2)$.

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 $\frac{s-1}{s}$ 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 [8], [9], 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 $\{\tau_1, \tau_2, \cdots, \tau_L\}$, thereafter rounding the deadline of a task down to the closest τ_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 [16] that is also designed via the general dynamic programming procedure. However, before our work, how to enable this extension was an open problem as pointed out in [3], [4]. We will show that this is mainly due to the conceptual lack of the optimal state of machines being utilized by malleable tasks with deadlines and the lack of an algorithm that achieves the optimal schedule. In contrast, the optimal resource utilization state in the single machine case can be defined much more easily and be achieved by the existing EDF algorithm. The core result of this paper fills the above gap and enables the design of the DP algorithm. The way of applying the core result to design a greedy algorithm is even less obvious since in the single machine case there is no corresponding algorithm to hint its role in the design of a greedy algorithm. So, new insights (into the above basic question) are needed to enable this new application and will be obtained through a complex algorithmic analysis.

The remainder of this paper is organized as follows. In Section II, we introduce the model of machines and tasks and the scheduling objectives considered in this paper. In Section III, we identify what the optimal resource utilization state is and propose such a scheduling algorithm that achieves the optimal state. In Section IV, we show three applications of the results in Section III in different algorithmic design techniques and scheduling objectives. Finally, we conclude in Section V. Due to space limitation, the proofs of all results are omitted and can be found in our technical report [18].

II. Model

There are C identical machines and a set of tasks $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$. 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: $\{1, 2, \cdots, d\}$, where $d = \max_{T_i \in \mathcal{T}} d_i$ 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 $y_i : [1, d_i] \to \{0, 1, 2, \cdots, k_i\}$, 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 $D_i, d_i \in \mathcal{Z}^+$ for all $T_i \in \mathcal{T}$. The value v_i of a task T_i can be obtained only if it is *fully allocated* by the deadline, i.e., $\sum_{t \le d_i} y_i(t) \ge D_i$, and partial execution of a task yields no value. Let $k = \max_{T_i \in \mathcal{T}} k_i$ be the maximum parallelism bound. For the system of C machines, denote by $W(t) = \sum_{i=1}^{n} y_i(t)$ the workload of the system at time slot t; and by $\overline{W}(t) = C - W(t)$ its complementary, i.e., the amount of available machines at time t. We call time t to be *fully utilized* (resp. saturated) if $\overline{W}(t) = 0$ (resp. $\overline{W}(t) < k$), and to be *not fully utilized* (resp. unsaturated) otherwise, i.e., if $\overline{W}(t) > 0$ (resp. $\overline{W}(t) \ge k$). In addition, the tasks tend to be recurring in practice, e.g., they are scheduled periodically on an hourly basis [9]. 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 [14].

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

- *social welfare maximization:* choose an appropriate subset $\mathcal{S} \subseteq \mathcal{T}$ and produce a feasible schedule of $\mathcal{S}$ so as to maximize the social welfare $\sum_{T_i \in \mathcal{S}} v_i$ (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 $\mathcal{T}$ on C machines.

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., $W(t) \le C$.

Additional notation. We now introduce more concepts that will facilitate the subsequent algorithm analysis. We will denote by $[l]$ and $[l]^+$ the sets $\{0, 1, \cdots, l\}$ and $\{1, 2, \cdots, l\}$. Let $len_i = \lceil D_i/k_i \rceil$ denote the *minimal length of execution time* of T_i. Given a set of tasks $\mathcal{T}$, the deadlines d_i of all tasks $T_i \in \mathcal{T}$ constitute a finite set $\{\tau_1, \tau_2, \cdots, \tau_L\}$, where $L \le n$, $\tau_1, \cdots, \tau_L \in \mathcal{Z}^+$, and $0 = \tau_0 < \tau_1 < \cdots < \tau_L = d$. Let $\mathcal{D}_i = \{T_{i,1}, T_{i,2}, \cdots, T_{i,n_i}\}$ denote the set of tasks with deadline τ_i ($i \in [L]^+$). Let $\mathcal{D}_{i,j}$ denote the set of tasks with deadline τ_i and the minimal length of execution time in $(\tau_i - \tau_{i-j+1}, \tau_i - \tau_{i-j}]$. Denote by $s_i = \frac{d_i}{len_i}$ the slackness of 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 $s = \min_{T_j \in \mathcal{T}} s_j$ be the slackness of the least flexible task ($s \ge 1$). Denote by $v_i' = \frac{v_i}{D_i}$ 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 $\mathcal{S}$, we use its capital S to denote the total demand of the tasks in $\mathcal{S}$. Let $D = \max_{T_i \in \mathcal{T}} \{D_i\}$ be the demand of the largest task.

III. Optimal Schedule

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

A. Optimal Resource Utilization State

For the tasks in our model, the deadline d_i decides the time interval in which a task can utilize the machines, and the parallelism bound k_i restricts that T_i can utilize at most

Algorithm 1: LDF($\mathcal{S}$)

Output: A feasible allocation of machines to a set of tasks $\mathcal{S}$

for $m \leftarrow L$ **to** 1 **do**
  while $\mathcal{S}_m \neq \emptyset$ **do**
    Get T_i from $\mathcal{S}_m$
    Allocate-B(i)
    $\mathcal{S}_m \leftarrow \mathcal{S}_m - \{T_i\}$

Algorithm 2: Allocate-B(i)

Fully-Utilize(i)
Fully-Allocate(i)
AllocateRLM(i, 1)

k_i machines at every time slot in $[1, d_i]$. Let $\mathcal{S} \subseteq \mathcal{T}$, and denote $\mathcal{S}_i = \mathcal{S} \cap \mathcal{D}_i$ and $\mathcal{S}_{i,j} = \mathcal{S} \cap \mathcal{D}_{i,j}$ ($i \in [L]^+$, $j \in [i]^+$). Let $\lambda_m(\mathcal{S}) = \sum_{l=L-m+1}^{L} \{\sum_{j=1}^{l-L+m} S_{l,j} + \sum_{j=l-L+m+1}^{l} \sum_{T_i \in \mathcal{S}_{l,j}} k_i(\tau_l - \tau_{L-m})\}$ for all $m \in [L]^+$. Here, a task $T_i \in \mathcal{S}_{l,j}$ ($j \in \{1, \cdots, l-L+m\}$) can utilize and only need D_i resources in $[\tau_{L-m}+1, d]$ and a task $T_i \in \mathcal{S}_{l,j}$ ($j \in \{l-L+m+1, \cdots, l\}$) can utilize at most $k_i(\tau_l - \tau_{L-m})$ resources in $[\tau_{L-m}+1, d]$ with the constraints of the deadline and parallelism bound. Hence, without the capacity constraint, $\lambda_m(\mathcal{S})$ represents the maximal workload of $\mathcal{S}$ that can be executed in the time slot interval $[\tau_{L-m}+1, d]$.

Lemma 3.1: Let $\lambda_m^C(\mathcal{S}) = \lambda_{m-1}^C(\mathcal{S}) + \min\{\lambda_m(\mathcal{S}) - \lambda_{m-1}^C(\mathcal{S}), C(\tau_{L-m+1} - \tau_{L-m})\}$, where $\lambda_0^C(\mathcal{S}) = 0$ and $m \in [L]^+$. $\lambda_m^C(\mathcal{S})$ is the maximal (optimal) workload of $\mathcal{S}$ that could could be executed in time slot interval $[\tau_{L-m}+1, d]$ on C machines ($m \in [L]^+$).

According to Lemma 3.1, for all $m \in [L]^+$, if a set of malleable tasks with deadlines $\mathcal{S}$ utilizes $\lambda_m^C(\mathcal{S})$ resources in every $[\tau_{L-m}+1, d]$ on C machines, *an optimal state* is achieved *in which C machines are optimally utilized by this set of tasks*. Let $\mu_m^C(\mathcal{S}) = S - \lambda_{L-m}^C(\mathcal{S})$ denote the remaining workload that needs to be executed after $\mathcal{S}$ has optimally utilized C machines in the time interval $[\tau_m+1, d]$.

Lemma 3.2: A *necessary condition* for the existence of a feasible schedule for a set of malleable batch tasks with deadlines $\mathcal{S}$ is the following:

$$\mu_m^C(\mathcal{S}) \leq C\tau_m, \text{ for all } m \in [L].$$

We refer to the necessary condition in Lemma 3.2 as *boundary condition*.

B. Scheduling Algorithm

In this section, we introduce the proposed optimal scheduling algorithm LDF($\mathcal{S}$) (latest deadline first), presented as Algorithm 1.

LDF($\mathcal{S}$). In this algorithm, we consider the tasks in the non-increasing order of the deadlines. For every task T_i being considered, the resource allocation algorithm Allocate-B(i) is called, presented as Algorithm 2, to allocate D_i resource to a task T_i without violating the constraints from deadline and parallelism bound. Upon every completion of Allocate-B($\cdot$), it achieves a special optimal resource utilization state such that the currently fully allocated tasks $\mathcal{S}' \subseteq \mathcal{S}_L \cup \cdots \cup \mathcal{S}_m$ will have been allocated $\lambda_l^C(\mathcal{S}')$ resource on C machines in $[\tau_{L-l}+1, d]$, $l \in [L]^+$. Such state also ensures that when the next task T_j is considered, Allocate-B(j) is able to fully allocate D_j resource to it iff $\mathcal{S}' \cup \{T_j\}$ satisfies the boundary condition. If so, LDF($\mathcal{S}$) will give a feasible schedule for a set of tasks $\mathcal{S}$ only if $\mathcal{S}$ satisfies the boundary condition.

Algorithm 3: Fully-Utilize(i)

for $t \leftarrow d_i$ **to** 1 **do**
  $y_i(t) \leftarrow \min\{k_i, D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t}), \overline{W}(t)\}$

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, η_1) 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 T_i 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 T_i at every time slot t is done from the deadline towards earlier time slots, and T_i is allocated $\min\{k_i, D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t}), \overline{W}(t)\}$ machines at t. $\min\{k_i, D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t}), \overline{W}(t)\}$ is the maximal amount of machines it can or need to utilize at t.

Lemma 3.3: Upon completion of Fully-Utilize(i), if $\overline{W}(t) > 0$, Fully-Utilize(i) will allocate $\min\{k_i, D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t})\}$ machines to T_i; further, if $D_i - \sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) > 0$, we have $y_i(t) = k_i$.

When a task T_i has been allocated by Fully-Utilize(i), we cannot guarantee that it is fully allocated D_i resources. Hence, more algorithms are needed: Fully-Allocate(i) and AllocateRLM(i, η_1).

Routine(Δ, η_1, η_2). We first introduce an algorithm Routine(Δ, η_1, η_2) that will be called in these two algorithms, also presented as Algorithm 4. Routine($\cdot$) focuses on the resource allocation at a single time slot t, and aims to increase the number of available machines $\overline{W}(t)$ at t to Δ by transferring the allocation of other tasks to an earlier time slot. Here, parameters η_1 and η_2 decide the exit condition of the loop in Routine($\cdot$). In Allocate-B(i), $\eta_1 = 1$. When Routine($\cdot$) is called in Fully-Allocate(i) (resp. AllocateRLM(i, η_1)), $\eta_2 = 0$ (resp. $\eta_2 = 1$).

In particular, there is a loop in Routine($\cdot$) and at the beginning of every loop iteration, let t' be the time slot earlier than but closest to t such that $\overline{W}(t') > 0$, and Routine($\cdot$) 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 $\overline{W}(t)$ is Δ, and (ii)

Algorithm 4: Routine(Δ, η_1, η_2)

while $\overline{W}(t) < \Delta$ **do**
  $t' \leftarrow$ the time slot earlier than and closest to t so that $\overline{W}(t') > 0$
  if $\eta_1 = 1$ **then**
    if *there exists no such* t' **then**
      break
  else
    if $t' \leq t_m^{th}$, *or there exists no such* t' **then**
      break
  if $\eta_2 = 1 \wedge \sum_{\bar{t}=1}^{t'-1} y_i(t') \leq \overline{W}(t)$ **then**
    break
  let i' be a job such that $y_{i'}(t) > y_{i'}(t')$
  $y_{i'}(t) \leftarrow y_{i'}(t) - 1$, $y_{i'}(t') \leftarrow y_{i'}(t') + 1$

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 $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) \leq \overline{W}(t)$. 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-Allocate(i) and AllocateRLM(i, η_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 $\Omega = D_i - \sum_{\bar{t}=1}^{d_i} y_i(\bar{t})$ 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 Δ by calling Routine(Δ, 1, 0). Subsequently, the algorithm updates Ω to be $\Omega - \overline{W}(t)$, and allocates the current available machines $\overline{W}(t)$ at t to T_i. Here, upon completion of its loop iteration of Fully-Allocate(i) at t, $\overline{W}(t) = 0$ if Fully-Allocate(i) has increased the allocation of T_i at t; in this case we will also see that $\overline{W}(t) = 0$ 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 $\overline{W}(t) > 0$ upon its completion, we also have that $\overline{W}(t) > 0$ 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 $\overline{W}(t) = 0$ 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 $\overline{W}(t') > 0$ in the loop of Routine($\cdot$), 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 $W(t) - y_i(t) > W(t') - y_i(t')$ and 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($\cdot$), $\overline{W}(t)$ becomes greater than 0 but $\overline{W}(t) < \Delta \leq k_i - y_i(t)$. We still have $W(t) - y_i(t) = C - \overline{W}(t) - y_i(t) > W(t') - k_i = W(t') - y_i(t')$ and such $T_{i'}$ can still be found.

Algorithm 5: Fully-Allocate(i)

$t \leftarrow d_i$, $\Omega \leftarrow D_i - \sum_{\bar{t} \leq d_i} y_i(\bar{t})$
while $\Omega > 0$ **do**
  $\Delta \leftarrow \min\{k_i - y_i(t), \Omega\}$
  Routine(Δ, 1, 0)
  $\Omega \leftarrow \Omega - \overline{W}(t)$, $y_i(t) \leftarrow y_i(t) + \overline{W}(t)$
  $t \leftarrow t - 1$

Algorithm 6: AllocateRLM(i, η_1)

$t \leftarrow d_i$
while $\sum_{\bar{t}=1}^{t-1} y_i(\bar{t}) > 0$ **do**
  $\Delta \leftarrow \min\{k_i - y_i(t), \sum_{\bar{t}=1}^{t-1} y_i(\bar{t})\}$
  Routine(Δ, η_1, 1)
  $\theta \leftarrow \overline{W}(t)$, $y_i(t) \leftarrow y_i(t) + \overline{W}(t)$
  let t'' be such a time slot that $\sum_{\bar{t}=1}^{t''-1} y_i(\bar{t}) < \theta$ and $\sum_{\bar{t}=1}^{t''} y_i(\bar{t}) \geq \theta$
  $\theta \leftarrow \theta - \sum_{\bar{t}=1}^{t''-1} y_i(\bar{t})$, $y_i(t'') \leftarrow y_i(t'') - \theta$
  for $\bar{t} \leftarrow 1$ **to** $t'' - 1$ **do**
    $y_i(\bar{t}) \leftarrow 0$
  $t \leftarrow t - 1$

AllocateRLM(i, η_1). Without changing the total allocation of T_i in $[1, d_i]$, AllocateRLM(i, η_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, η_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 $\sum_{\bar{t}=1}^{t-1} y_i(\bar{t})$ of T_i in $[1, t-1]$ is greater than 0, AllocateRLM($\cdot$) begins its loop iteration at t. Let $\Delta = \min\{k_i - y_i(t), \sum_{\bar{t}=1}^{t-1} y_i(\bar{t})\}$ and Δ is the maximal extra machines that T_i can utilize at t. If $\Delta > 0$, we enter Routine(Δ, η_1, η_2). Here, $\Delta > 0$ also means $y_i(t) < k_i$. When Routine($\cdot$) stops, we have that the number of available machines $\overline{W}(t)$ at t is no more than Δ. Let u_t be the last time

slot t' considered in the loop of Routine($\cdot$) for t such that the total allocation at $t^{'}$ has been increased. In a different case than the current state here, AllocateRLM($\cdot$) does nothing and take no effect on the allocation of T_i at t; then set $u_t = u_{t+1}$ if $t < d_i$ and $u_t = d_i$ if $t = d_i$. Then, AllocateRLM(i, η_1) decreases the current allocation of T_i at the earliest time slots in $[1, u_t - 1]$ to 0 (and by $\overline{W}(t)$), and accordingly increases the allocation of T_i at t by $\overline{W}(t)$. Here, upon completion of the loop iteration of AllocateRLM($\cdot$) at t, $\overline{W}(t) = 0$ if AllocateRLM($\cdot$) has taken an effect on the allocation of T_i at t; in this case $\overline{W}(t)$ also equals to 0 just before the execution of this loop iteration at t. Here, when AllocateRLM($\cdot$) is called in Allocate-B(i), $\eta_1 = \eta_2 = 1$. The reason for the existence of $T_{i'}$ is similar to but more complex than the case of Fully-Allocate(i), and is explained.

Difference of Routine($\cdot$) and GreedyRTL. The operations in Routine($\cdot$) 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 [18] 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 [18], GreedyRTL achieves a resource utilization of $\min\{\frac{s-1}{s}, \frac{C-k+1}{C}\}$ due to its allocation condition, which is not optimal.

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

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

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

In other words, if LDF($\mathcal{S}$) cannot produce a feasible schedule for a set of tasks $\mathcal{S}$, 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 social welfare maximization problem in [3], [4], giving the best possible greedy algorithm and the first exact dynamic programming algorithm. We also show its direct applications to the machine minimization problem.

A. Greedy Algorithm

Generic algorithm and its bound. 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 [19], [20]: 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 GREEDY of algorithms that operate as follows: (i) considers tasks in the non-increasing order of the marginal value; and (ii) let $\mathcal{A}$ denote the set of the tasks that have been accepted so far, and, for a task T_i being considered,

Algorithm 7: GreedyRLM

Input : n jobs with $type_i = \{v_i, d_i, D_i, k_j\}$
Output: A feasible allocation of resources to jobs

initialize: $y_i(t) \leftarrow 0$ for all $T_i \in \mathcal{T}$ and $1 \le t \le T$, $m = 0$, $t_m^{th} = 0$;
sort jobs in the non-increasing order of the marginal values: $v_1' \ge v_2' \ge \cdots \ge v_n'$;
$i \leftarrow 1$;
while $i \le n$ **do**
  if $\sum_{t \le d_i} \min\{\overline{W}(t), k_i\} \ge D_i$ **then**
    Allocate-A(i); // in the $(m+1)$-th phase
  else
    if T_{i-1} *has ever been accepted* **then**
      $m \leftarrow m + 1$; // in the m-th phase, the allocation to $\mathcal{A}_m$ was completed; the first rejected task is $T_{j_m} = T_i$
    while $\sum_{t \le d_{i+1}} \min\{\overline{W}(t), k_{i+1}\} < D_{i+1}$ **do**
      $i \leftarrow i + 1$;
    /* the last rejected task is $T_{i_{m+1}-1} = T_i$ and $\mathcal{R}_m = \{T_{j_m}, \cdots, T_{i_{m+1}-1}\}$ */
    if $c_m \ge c_m'$ **then**
      $t_m^{\text{th}} \leftarrow c_m$;
    else
      set t_m^{th} to time slot just before the first time slot t with $\overline{W}(t) > 0$ after c_m or to c_m' if there is no time slot t with $\overline{W}(t) > 0$ in $[c_m, c_m']$;
  $i \leftarrow i + 1$;

Algorithm 8: Allocate-A(i)

Fully-Utilize(i)
AllocateRLM(i, 0)

it is accepted and fully allocated *iff* there exists a feasible schedule for $\mathcal{A} \cup \{T_i\}$. In the following, we refer to the generic algorithm in GREEDY as *Greedy*.

Proposition 4.1: The best performance guarantee that a greedy algorithm in GREEDY can achieve is $\frac{s-1}{s}$.

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-1}, 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 $c_m = \max_{T_i \in \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_m} \{d_i\}$ 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 Greedy. While the tasks in $\mathcal{A}_m \cup \mathcal{R}_m$ are being considered, we refer to Greedy as being in the m-th phase. Before the execution of Greedy, we refer to it as being in the 0-th 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 $D_{m,i}^{[t_1,t_2]} = \sum_{t=t_1}^{t_2} y_i(t)$ to describe the current total allocation to T_i over $[t_1, t_2]$. After the completion of Greedy, we also define $D_{K+1,i}^{[t_1,t_2]} = \sum_{t=t_1}^{t_2} y_i(t)$ to describe the final total allocation to T_i over $[t_1, t_2]$. We further define $T_{K+1,i}^{[t_1,t_2]}$ as an *imaginary task* with characteristics $\{v_{K+1,i}^{[t_1,t_2]}, D_{K+1,i}^{[t_1,t_2]}, d_{K+1,i}^{[t_1,t_2]}, k_i\}$, where $v_{K+1,i}^{[t_1,t_2]} = v_i D_{K+1,i}^{[t_1,t_2]}/D_i$, $d_{K+1,i}^{[t_1,t_2]} = \min\{t_2, d_i\}$.

Algorithmic analysis: features and theorem. We now define two features of the resource allocation structure related to the accepted tasks of Greedy. In fact, if the resource allocation structure of Greedy satisfies such features, its performance guarantee can be deduced immediately.

Upon completion of the m-th phase of Greedy, we define the *threshold* parameter t_m^{th} as follows. If $c_m \geq c'_m$, then set $t_m^{\text{th}} = c_m$. If $c_m < c'_m$, then set t_m^{th} to a certain time slot in $[c_m, c'_m]$. We emphasize here that $d_i \leq t_m^{th}$ for all $T_i \in \cup_{j=1}^m \mathcal{R}_j$ hence the allocation to the tasks of $\cup_{i=1}^m \mathcal{R}_i$ in $[t_m^{th}+1, T]$ is ineffective and yields no value due to the constraint from the deadline. For ease of exposition, we let $t_0^{th} = 0$ and $t_{K+1}^{th} = T$. 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 $\cup_{j=1}^m \mathcal{A}_j$ in $[1, t_m^{th}]$ is at least r, i.e., $\sum_{T_i \in \cup_{j=1}^m \mathcal{A}_j} D_{K+1,i}^{[1,t_m^{th}]} / (C \cdot t_m^{th}) \geq r$.

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

Feature 4.2: $[t_m^{\text{th}} + 1, t_{m+1}^{\text{th}}]$ is optimally utilized by $\{T_{K+1,i}^{[1,t_{m+1}^{th}]} \mid T_i \in \cup_{j=1}^m \mathcal{A}_j\}$.

Theorem 4.1: If 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 $\mathcal{A}_0 = \mathcal{R}_0 = \mathcal{A}_{K+1} = \mathcal{R}_{K+1} = \emptyset$.

Best possible greedy algorithm. We now introduce the executing process of the greedy algorithm 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* $\sum_{t \leq d_i} \min\{\overline{W}(t), k_i\} \geq D_i$, call Allocate-A($i$) to make T_i fully allocated. Here, Routine(Δ, 0, 1) exits only if in a loop iteration one of the following conditions is satisfied: (1) the number of current available machines $\overline{W}(t)$ is Δ, (2) there exists no such t', and (3) there exists such t' but either $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) \leq \overline{W}(t)$ or $t' \leq t_m^{th}$. The existence of $T_{i'}$ is also explained in our technical report [18].

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

Proposition 4.2: GreedyRLM gives an $\frac{s-1}{s}$-approximation to the optimal social welfare with a time complexity of $\mathcal{O}(n^2)$.

B. Dynamic Programming Algorithm

In this section, we show the application of the dynamic programming technique to the social welfare maximization problem.

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 $\mathcal{S}$ of tasks in $\mathcal{D}_1 \cup \cdots \cup \mathcal{D}_L$ 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 $\mathcal{F} \subseteq \mathcal{T}$ and we define a L-dimensional vector

$$H(\mathcal{F}) = (\lambda_1^C(\mathcal{F}) - \lambda_0^C(\mathcal{F}), \cdots, \lambda_L^C(\mathcal{F}) - \lambda_{L-1}^C(\mathcal{F})),$$

where $\lambda_m^C(\mathcal{F}) - \lambda_{m-1}^C(\mathcal{F})$, $m \in [L]^+$, denotes the optimal resource that $\mathcal{F}$ can utilize on C machines in the segmented timescale $[\tau_{L-m}+1, \tau_{L-m+1}]$ after $\mathcal{F}$ has utilized $\lambda_{m-1}^C(\mathcal{F})$ resource in $[\tau_{L-m+1}+1, \tau_L]$. Let $v(\mathcal{F})$ denote the total value of the tasks in $\mathcal{F}$ and then we introduce the notion of one pair $(\mathcal{F}, v(\mathcal{F}))$ *dominating* another $(\mathcal{F}', v(\mathcal{F}'))$ if $H(\mathcal{F}) = H(\mathcal{F}')$ and $v(\mathcal{F}) \geq v(\mathcal{F}')$, that is, the solution to our problem indicated by $(\mathcal{F}, v(\mathcal{F}))$ uses the same amount of resources as $(\mathcal{F}', v(\mathcal{F}'))$, but obtains at least as much value.

We now give the general DP procedure DP($\mathcal{T}$) [21]. Here, we iteratively construct the lists $A(j)$ for all $j \in [n]^+$. Each

$A(j)$ is a list of pairs $(\mathcal{F}, v(\mathcal{F}))$, in which $\mathcal{F}$ is a subset of $\{T_1, T_2, \cdots, T_j\}$ satisfying the boundary condition and $v(\mathcal{F})$ is the total value of the tasks in $\mathcal{F}$. Each list only maintains all the dominant pairs. Specifically, we start with $A(1) = \{(\emptyset, 0), (\{T_1\}, v_1)\}$. For each $j = 2, \cdots, n$, we first set $A(j) \leftarrow A(j-1)$, and for each $(\mathcal{F}, v(\mathcal{F})) \in A(j-1)$, we add $(\mathcal{F} \cup \{T_j\}, v(\mathcal{F} \cup \{T_j\}))$ to the list $A(j)$ if $\mathcal{F} \cup \{T_j\}$ satisfies the boundary condition. We finally remove from $A(j)$ all the dominated pairs. DP($\mathcal{T}$) will select a subset $\mathcal{S}$ of $\mathcal{T}$ from all pairs $(\mathcal{F}, v(\mathcal{F})) \in A(n)$ so that $v(\mathcal{F})$ is maximal.

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

Discussion. As in the knapsack problem [21], to construct the algorithm DP($\mathcal{T}$), 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 $\mathcal{T}$, the minimal number of machines needed to produce a feasible schedule of $\mathcal{T}$ 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 3.1 and Lemma 3.2:

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

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 $\mathcal{O}(n^2)$ that achieves an approximation ratio of $\frac{s-1}{s}$, (ii) the first dynamic programming algorithm for social welfare maximization with a polynomial time complexity of $\mathcal{O}(\max\{nd^L C^L, n^2\})$, (iii) the first exact algorithm for machine minimization with a polynomial time complexity of $\mathcal{O}(n^2)$. 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 malleable tasks on identical machines respectively to the case with release time and to the case in which each task consists 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

[1] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. "Bridging the Tenant-Provider Gap in Cloud Services." In Proceedings of the 3rd ACM Symposium on Cloud Computing. ACM, 2012.

[2] V. Ishakian, R. Sweha, A. Bestavros, and J. Appavoo. "CloudPack: Exploiting Workload Flexibility through Rational Pricing." In Proceedings of the ACM/IFIP/USENIX 13th International Middleware Conference. Springer, 2012.

[3] N. Jain, I. Menache, J. Naor, and J. Yaniv. "A Truthful Mechanism for Value-Based Scheduling in Cloud Computing." In Proceedings of the 4th International Conference on Algorithmic Game Theory. Springer, 2011.

[4] N. Jain, I. Menache, J. Naor, and J. Yaniv. "Near-Optimal Scheduling Mechanisms for Deadline-Sensitive Jobs in Large Computing Clusters." In Proceedings of the 24th ACM symposium on Parallelism in Algorithms and Architectures. ACM, 2012.

[5] B. Lucier, I. Menache, J. Naor, and J. Yaniv. "Efficient Online Scheduling for Deadline-Sensitive Jobs." In Proceedings of the 25th ACM symposium on Parallelism in Algorithms and Architectures. ACM, 2013.

[6] N. Jain, I. Menache, and O. Shamir. "On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud." In the 11th International Conference on Autonomic Computing. USENIX Association, 2014.

[7] Y. Azar, I. Kalp-Shaltiel, B. Lucier, I. Menache, J. Naor, and J. Yaniv. "Truthful Online Scheduling with Commitments." In Proceedings of the 16th ACM conference on Economics and Computation. ACM, 2015.

[8] P. Bodik, I. Menache, J. Naor, and J. Yaniv. "Brief Announcement: Deadline-Aware Scheduling of Big-Data Processing Jobs." In Proceedings of the 26th ACM symposium on Parallelism in Algorithms and Architectures. ACM, 2014.

[9] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and Rodrigo Fonseca. "Jockey: Guaranteed Job Latency in Data Parallel Clusters." In Proceedings of the 7th ACM European Conference on Computer Systems. ACM, 2012.

[10] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan et al. "Apache Hadoop goes realtime at Facebook." In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. ACM, 2011.

[11] H. Hu, Y. Wen, T.S. Chua and X. Li. "Toward Scalable Systems for Big Data Analytics: A Technology Tutorial." IEEE Access, vol. 2, 2014, pp.652-687.

[12] V. Nagarajan, J. Wolf, A. Balmin, and K. Hildrum. "FlowFlex: Malleable Scheduling for Flows of MapReduce Jobs." In Proceedings of the ACM/IFIP/USENIX 14th International Middleware Conference. Springer, 2013.

[13] J. Wolf, Z. Nabi, V. Nagarajan, R. Saccone, R. Wagle, K. Hildrum, E. Pring, and K. Sarpatwar. "The X-flex Cross-Platform Scheduler: Who's the Fairest of Them All?." In Proceedings of the ACM/IFIP/USENIX 13th International Middleware Conference, Industry Track. ACM, 2014.

[14] T. White. "Hadoop: The definitive guide." O'Reilly Media, Inc., 2012.

[15] D. Karger, C. Stein, and J. Wein. Scheduling Algorithms. In CRC Handbook of Computer Science. 1997.

[16] E. L. Lawler, "A Dynamic Programming Algorithm for Preemptive Scheduling of a Single Machine to Minimize the Number of Late Jobs." Annals of Operations Research 26.1 (1990): 125-133.

[17] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms. Kluwer Academic, 1998

[18] X. Wu and P. Loiseau. "Algorithms for Scheduling Deadline-Sensitive Malleable Tasks." arXiv Preprint arXiv:1501.04343v4, 2015.

[19] G. Brassard, and P. Bratley. Fundamentals of Algorithmics. Prentice-Hall, Inc., 1996.

[20] G. Even, Recursive greedy methods, in Handbook of Approximation Algorithms and Metaheuristics, T. F. Gonzalez, ed., CRC, Boca Raton, FL, 2007, ch. 5.

[21] D. P. Williamson and D. B. Shmoys. The Design of Approximation Algorithm. Cambridge University Press, 2011.