Competitive Online Adaptive Scheduling for Sets of Parallel Jobs with Fairness and Efficiency

Hongyang Sun[1], Wen-Jing Hsu[1], and Yangjie Cao[2]

[1]School of Computer Engineering, Nanyang Technological University, Singapore
[2]School of Software Engineering, Zhengzhou University, China
{sunh0007, hsu}@ntu.edu.sg, caoyj@zzu.edu.cn

Abstract: We study online adaptive scheduling for multiple sets of parallel jobs, where each set may contain one or more jobs with time-varying parallelism. This two-level scheduling scenario arises naturally when multiple parallel applications are submitted by different users or user groups in large parallel systems, where both user-level fairness and system-wide efficiency are of important concerns. To achieve fairness, we use the well-known equi-partitioning algorithm to distribute the available processors among the active job sets at any time. For efficiency, we apply a feedback-driven adaptive scheduler that periodically adjusts the processor allocations within each set by consciously exploiting the jobs' execution history. We show that our algorithm achieves asymptotically competitive performance with respect to the set response time, which incorporates two widely used performance metrics, namely, total response time and makespan, as special cases. Both theoretical analysis and simulation results demonstrate that our algorithm improves upon an existing scheduler that provides only fairness but lacks efficiency. Furthermore, we provide a generalized framework for analyzing a family of scheduling algorithms based on feedback-driven policies with provable efficiency. Finally, we consider an extended multi-level hierarchical scheduling model and present a fair and efficient solution that effectively reduces the problem to the two-level model.

Keywords: Adaptive scheduling, parallel jobs, set response time, multiprocessors, online algorithm, competitive analysis, hierarchical scheduling, fairness, efficiency

1 Introduction

1.1 Background

Scheduling parallel jobs on multiprocessor systems has been a fundamental area of research in computer science for decades [10, 14, 18]. As parallel systems are increasingly deployed to provide high-performance computing services, such as in cloud computing and the large-scale data centers, efficient scheduling on these platforms will play a more important role in boosting application performance and increasing system utilization.

Most parallel systems today are shared by multiple users, and a common scenario arises when each user submits several jobs to the system. A natural objective in this case is to achieve efficient execution of the jobs while at the same time offering a level of fairness among different users. In this paper, we consider such a scenario, in which a collection of parallel job sets needs to be scheduled on a multiprocessor system and each job set corresponds to the set of applications submitted by a particular user or user group. We are interested in the *response time* of a job set, which is the duration between when the job set is submitted and when all jobs in the job set are completed. The objective is to minimize the sum of the response times of all job sets, or the *set response time*. As pointed out by Robert and Schabanel [19], the metric of set response time benchmarks both fairness and efficiency of a scheduling algorithm. In fact, it represents a more general performance measure that incorporates two widely used metrics, namely, *total response time* and *makespan*, as special cases. Suppose that each job set in the collection contains only a single job, the set response time becomes the total response time of all jobs in the collection. At the other extreme, if the collection contains only a single job set, the set response time is simply the makespan of all jobs. To schedule a collection of job sets, an algorithm needs to allocate processors at two separate levels, namely, the *job-set level* and the *job level*. In particular, it needs to first specify the number of processors allocated to each job set, and then decides the processor allocation for the jobs within each job set. Such a two-level scheduling model is considerably more challenging compared to the traditional single-level scheduling [8, 7, 1, 11, 25], where an algorithm only needs to decide the processor

allocation for a flat collection of jobs.

We consider parallel jobs with time-varying parallelism profile, which is commonly observed in many applications that go through different phases in their executions. Each phase of a job is specified by an amount of work to be done and a speedup function, which we assume in this paper is linear up to a phase-dependent maximum. (See Section 2.1 for the detailed job model.) Moreover, the parallel jobs are assumed to be *malleable* [10] in nature, that is, they can adjust to the changing processor allocations at runtime. Such malleability is enabled by the more flexible runtime systems [30, 21, 6, 23] that have emerged in the last decade as well as the state-of-the-art virtual machine (VM) technology [2, 16, 22] that makes adaptive scheduling possible with little or negligible overhead. In contrast to *static scheduling* [14], which allocates a fixed set of processors to a job throughout its execution, *adaptive scheduling* can benefit from the time-varying characteristic of the jobs' resource requirements and hence appears to be a more promising approach to scheduling jobs in modern parallel systems. We adopt the *online non-clairvoyant* scheduling model [17, 11], which requires an algorithm to make all scheduling decisions in an online manner without any knowledge of the jobs' future characteristics, such as their release time, remaining work and parallelism profile. This is a natural assumption since such information is generally not available to the operating system schedulers. We measure the performance of an online adaptive scheduler using the standard *competitive analysis* [3], which compares its set response time with that of an optimal offline scheduler.

1.2 Related Work

A well-known online adaptive scheduler is Equi-partitioning (EQUI) [29], which at any time divides the total number of processors evenly among all active jobs in the system. This algorithm, although simple, is able to ensure fairness by automatically adjusting the processor allocations whenever a new job is admitted into the system or an existing job is completed and leaves the system. In fact, such a simple notion of fairness is sufficient to guarantee satisfying performance when each user submits only one job. In particular, Edmonds et al. [8] showed that EQUI is $(2+\sqrt{3})$-competitive with respect to the total response time of all jobs if they are released at the same time. Using *resource augmentation analysis* [13], Edmonds [7] also showed that EQUI is $(2+4/\epsilon)$-competitive for arbitrary released jobs with processors whose speed is $2+\epsilon$ times faster than the optimal, for any $\epsilon > 0$.[1] However, despite its good performance for the total response time, EQUI fares poorly in terms of the makespan, which to a certain extent reflects the system efficiency when there is only one user in the system. Since EQUI does not consider how efficiently each job is able to utilize the allocated processors, it may under-utilize the resources especially when different jobs can have very different processor requirements. Specifically, Robert and Schabanel [19] showed that EQUI is $\Theta(\frac{\ln n}{\ln \ln n})$-competitive with respect to the makespan even if all jobs are batch released, where n is the total number of jobs submitted to the system.

It turns out that both user-level *fairness* and system-wide *efficiency* are critical when minimizing the set response time for a collection of job sets. In [19], Robert and Schabanel applied EQUI to both levels by equally dividing the total number of processors among the active job sets and within each job set equally dividing the allocated processors among its active jobs. They showed that the resulting algorithm EQUI$\circ$EQUI has a competitive ratio of $(2+\sqrt{3}+o(1))\frac{\ln n}{\ln \ln n}$ with respect to the set response time when all job sets are batch released, where n is the maximum number of jobs in any job set. The result suggests that the set response time ratio of a scheduling algorithm actually encompasses the total response time ratio and the makespan ratio of the corresponding algorithms at the job-set level and the job level, respectively. Hence, it is important to retain both fairness and efficiency in order to achieve satisfying performance for this general scheduling metric.

To improve the system efficiency, feedback-driven adaptive schedulers [1, 11, 25] were recently proposed. Unlike EQUI, which obliviously allocates processors to the jobs regardless of their actual resource requirements, feedback-driven algorithms periodically adjust processors among the jobs by consciously exploiting the jobs' execution history. In particular, Agrawal et al. [1] introduced the A-GREEDY scheduler that periodically collects the resource utilization of each job, and based on this information estimates the job's future processor requirement. It has been shown that A-GREEDY wastes at most a constant fraction of a job's allocated processors, and thus achieves efficient processor utilization [1]. Furthermore, by combining A-GREEDY with a conservative resource allocator, such as Dynamic Equi-partitioning (DEQ) [15], He et al. [11] showed that the feedback-driven algorithm AGDEQ is asymptotically $O(1)$-competitive with respect to the makespan regardless of the number of jobs in the system. Recently, Sun et al. [25] proposed another feedback-driven adaptive scheduler ACDEQ, which uses a control-theoretic approach to estimate the jobs' processor requirements and it has been shown to have better feedback stability and system efficiency.

[1] Edmonds and Pruhs [9] recently proposed a variant of the EQUI scheduler, called Latest Arrival Processor Sharing (LAPS), which at any time shares the total number of processors evenly among the β fraction of the active jobs with the latest release time, for any $0 < \beta \leq 1$. They showed that by varying the parameter β, LAPS provides a tradeoff between the augmented processor speed $s = 1+\beta+\epsilon$ and the competitive ratio $4s/(\beta\epsilon)$, for any $\epsilon > 0$.

1.3 Our Contributions

Aiming at both user-level fairness and system-wide efficiency, we bring together the benefit of the EQUI algorithm [29] and that of the feedback-driven scheduler AGDEQ [11], and propose a both fair and efficient online adaptive algorithm EQUI∘AGDEQ for scheduling any collection of job sets. We show that the set response time ratio of our algorithm indeed combines the total response time and the makespan ratios of the respective algorithms in a non-trivial manner. Our first contribution is to bound the asymptotic competitive ratio (see Section 2.2 for the detailed definition) of EQUI∘AGDEQ, which is summarized in the following.

- EQUI∘AGDEQ is $O(1)$-competitive in the asymptotic sense with respect to the set response time when all job sets are batch released. The exact ratio depends on the constant parameters of the AGDEQ algorithm, and it is formally stated in Theorem 1 (Section 4.2). This result improves the competitive ratio of $\Theta(\frac{\ln n}{\ln \ln n})$ achieved by the EQUI∘EQUI algorithm [19] for sufficiently large jobs, where n is the maximum number of jobs in any job set. The improvement is a direct result that EQUI∘AGDEQ exhibits both fairness and efficiency while EQUI∘EQUI provides only fairness but lacks efficiency.
- EQUI∘AGDEQ is $O(1)$-competitive in the asymptotic sense with respect to the set response time for arbitrary released job sets if it is augmented with processors whose speed is $O(1)$ times faster than that of the optimal.[2] Again, the competitive ratio and the augmented speed depend on the constant parameters of the AGDEQ algorithm, and they are formally stated in Theorem 2 (Section 4.3). This result extends a similar performance bound obtained in [20] from jobs with specific parallelism structure, i.e., a sequential phase followed by a fully-parallelizable phase, to sufficiently large jobs with any parallelism profile.

Our second contribution is a generalized framework for analyzing the EQUI∘XY family of algorithms, which uses EQUI to distribute the processors among the job sets and any feedback-driven algorithm XY to schedule the jobs within each set. The performance of EQUI∘XY is then obtained by bounding the efficiency of the XY algorithm. We demonstrate the generality of the framework by applying it to the ACDEQ algorithm [25] as well as an algorithm that provides feedbacks based on the exact parallelism of the jobs at any time, assuming that such information can be made accessible to the online scheduler.

Furthermore, we consider a multi-level hierarchical scheduling problem, where arbitrary tree structures may exist between the source of the processors and the job sets as well as within each job set. Our third contribution is a fair and efficient solution to this problem using effective desire aggregation and processor allocation schemes. The solution reduces the multi-level scheduling problem to the two-level scheduling model. The reduced problem can then be solved by using our proposed algorithms with provable efficiency.

Finally, our last contribution is a simulation study that empirically evaluates the performance of the EQUI∘AGDEQ algorithm using synthetic parallel jobs with internal parallelism variations [4]. The simulation also compares EQUI∘AGDEQ with EQUI∘EQUI and the EQUI∘ACDEQ algorithm, which combines EQUI with the feedback-driven scheduler ACDEQ [25]. The results confirm that the algorithms based on feedback-driven schedulers indeed achieve better system efficiency and exhibit superior performance in terms of the set response time.

1.4 Organization of the Paper

The rest of this paper is organized as follows. Section 2 formally defines the two-level scheduling problem. Section 3 describes the EQUI∘AGDEQ algorithm, followed by its performance analysis for both batched and arbitrary released job sets in Section 4. Section 5 provides a generalized analysis framework and demonstrates its application. Section 6 extends the two-level scheduling model to a multi-level hierarchical scheduling problem and presents our solution. Section 7 presents some simulation results, and finally, Section 8 concludes the paper.

2 Problem Formulation

2.1 Job Model and Scheduling Model

We adopt the model in [24, 25, 26] to represent a parallel job that consists of a series of phases with different degrees of parallelism. Specifically, we consider a collection $\mathcal{J} = \{\mathcal{J}_1, \mathcal{J}_2, \cdots, \mathcal{J}_m\}$ of m job sets, which correspond to m different users. Each job set $\mathcal{J}_i = \{J_{i1}, J_{i2}, \cdots, J_{in_i}\}$ contains n_i jobs, which correspond to the applications submitted by the i-th user. Each job $J_{ij} = \langle J_{ij}^1, J_{ij}^2, \cdots, J_{ij}^{k_{ij}} \rangle$ contains k_{ij} phases, where each

[2] It is straightforward to observe that the LAPS scheduler proposed in [9] can be used at the job-set level to replace EQUI. The resulting algorithm will provide a similar tradeoff as in [9] between the achieved competitive ratio and the required speed augmentation.

phase J_{ij}^k has an amount of *work* $w_{ij}^k > 0$ and a *maximum parallelism* $h_{ij}^k \geq 1$. Suppose that at time t job J_{ij} is in its k-th phase and is allocated a_{ij}^k processors. Then its effective *speedup* is linear up to the phase's maximum parallelism, and is given by $\Gamma_{ij}^k(t) = \min\{a_{ij}^k, h_{ij}^k\}$. Hence, on processors of speed s for any $s > 0$, the *execution rate* of the job at time t is given by $s \cdot \Gamma_{ij}^k(t)$. The *span* l_{ij}^k of phase J_{ij}^k, which represents the amount of time to execute the phase with h_{ij}^k or more processors of unit speed, is therefore $l_{ij}^k = w_{ij}^k / h_{ij}^k$. The *work* $w(J_{ij})$ of job J_{ij} is given by $w(J_{ij}) = \sum_{k=1}^{k_{ij}} w_{ij}^k$, and the *span* $l(J_{ij})$ of the job is $l(J_{ij}) = \sum_{k=1}^{k_{ij}} l_{ij}^k$. Moreover, for each job set $\mathcal{J}_i$, we define its *set work* to be $w(\mathcal{J}_i) = \sum_{j=1}^{n_i} w(J_{ij})$ and define its *set span* to be $l(\mathcal{J}_i) = \max_{j=1\cdots n_i} l(J_{ij})$.

Given a total number P of processors, a scheduling algorithm needs to decide at any time t the processor allocation $a(\mathcal{J}_i, t)$ for each job set $\mathcal{J}_i$ as well as the processor allocation $a(J_{ij}, t)$ for each job J_{ij} within the job set. We require that the total processor allocation cannot exceed the total number of available processors, i.e., $\sum_{i=1}^{m} \sum_{j=1}^{n_i} a(J_{ij}, t) \leq P$. Let r_{ij} denote the *release time* of job J_{ij}, which is the time when the job is submitted, and let r_i denote the release time of job set $\mathcal{J}_i$. In this paper, we assume that all jobs in a job set are submitted at the same time, i.e., $r_{ij} = r_i$ for all $1 \leq j \leq n_i$. Moreover, if all job sets are submitted in a single batch, their release times are equal to 0. Otherwise, we can assume without loss of generality that the first submitted job set arrives at time 0. Let c_{ij}^k denote the completion time of the k-th phase of job J_{ij}, and let $c_{ij} = c_{ij}^{k_{ij}}$ denote the *completion time* of job J_{ij}. The completion time c_i of job set $\mathcal{J}_i$ is given by $c_i = \max_{j=1\ldots n_i} c_{ij}$. We also require that a valid schedule cannot begin to execute a phase of a job unless it has completed all its previous phases. To simplify analysis as in many previous work [7, 8, 11, 19, 20, 24, 25], we allow the processor allocation of a job to be fractional, and the processors can be reallocated among the sets and the jobs at any time without penalty. This can be achieved by the more flexible runtime systems [30, 21, 6, 23] and/or the state-of-the-art virtual machine technology [2, 16, 22].

2.2 Objective Function

The *response time* or *flow time* f_{ij} of job J_{ij} is the duration between the completion time and the release time of the job, i.e., $f_{ij} = c_{ij} - r_{ij}$, and the *response time* f_i of job set $\mathcal{J}_i$ is given by $f_i = c_i - r_i$. The *total response time* $F(\mathcal{J})$ of all jobs in $\mathcal{J}$ is $F(\mathcal{J}) = \sum_{i=1}^{m} \sum_{j=1}^{n_i} f_{ij}$ and the *makespan* $M(\mathcal{J})$ of the jobs is $M(\mathcal{J}) = \max_{i=1\cdots m, j=1\cdots n_i} c_{ij}$. Our objective is to minimize the total response time of all job sets, or the *set response time* $H(\mathcal{J})$, which is given by $H(\mathcal{J}) = \sum_{i=1}^{m} f_i$. A job J_{ij} is said to be *active* at time t if it has been released but not completed at t, i.e., $r_{ij} \leq t \leq c_{ij}$. A job set $\mathcal{J}_i$ is said to be *active* at time t if it contains at least one active job. An alternative expression for the set response time is thus given by $H(\mathcal{J}) = \int_0^\infty m_t dt$, where m_t denotes the number of active job sets at time t. As pointed out in [19], the set response time of a job set collection $\mathcal{J}$ has the following interesting property: if $\mathcal{J} = \{\mathcal{J}_1\}$ contains a single job set, then the set response time is simply the makespan of all jobs in $\mathcal{J}$; if $\mathcal{J} = \{\mathcal{J}_1, \mathcal{J}_2, \cdots, \mathcal{J}_m\}$ contains a collection of singleton job sets, where for each $i = 1 \cdots m$, we have $\mathcal{J}_i = \{J_{i1}\}$, then the set response time is the total response time of all jobs in $\mathcal{J}$. Hence, the objective of set response time represents a more general performance metric that incorporates both makespan and total response time as special cases.

We use *competitive analysis* [3] to compare the set response time of an online algorithm with that of an optimal offline scheduler. An online algorithm is said to be *c-competitive* if there exists a finite constant b such that the algorithm's set response time satisfies $H(\mathcal{J}) \leq c \cdot H^*(\mathcal{J}) + b$ for any job set collection $\mathcal{J}$, where $H^*(\mathcal{J})$ denotes the set response time of $\mathcal{J}$ under an optimal offline scheduler. Since an online algorithm does not have any prior knowledge about the jobs while the optimal offline scheduler knows everything in advance, it is generally not possible to get reasonably good competitive ratios when the job sets can have arbitrary release time [7, 20]. In this case, we use the *resource augmentation analysis* [13], which gives the online algorithm extra resources and in a sense limits the power of the adversary. An online algorithm is then said to be *s-speed c-competitive* if there exists a finite constant b such that the algorithm's set response time $H_s(\mathcal{J})$ on processors of speed s, where $s > 1$, satisfies $H_s(\mathcal{J}) \leq c \cdot H_1^*(\mathcal{J}) + b$ for any job set collection $\mathcal{J}$, where $H_1^*(\mathcal{J})$ denotes the set response time of $\mathcal{J}$ under an optimal offline scheduler using unit-speed processors. In both types of analysis, the competitive ratio c is said to be *strong* if $b = 0$. Otherwise, the competitive ratio is said to be achieved in the *asymptotic* sense.

2.3 Lower Bounds for Set Response Time

For any job set collection $\mathcal{J}$, we define its *total span* to be $l(\mathcal{J}) = \sum_{i=1}^{m} l(\mathcal{J}_i)$, and define its *squashed work* to be $\hat{w}(\mathcal{J}) = \frac{1}{P} \sum_{i=1}^{m} i \cdot w(\mathcal{J}_{\pi(i)})$, where $\pi(\cdot)$ denotes a permutation of the job sets sorted in non-increasing order of work, i.e., $w(\mathcal{J}_{\pi(1)}) \geq w(\mathcal{J}_{\pi(2)}) \geq \cdots \geq w(\mathcal{J}_{\pi(m)})$. Lemma 1 below states that the total span and the squashed work can be used as two lower bounds for the set response time of any job set collection, where the

latter only applies to batched job sets. In fact, these two lower bounds resemble the well-known lower bounds for the total response time of a single job set [8, 7, 11].

Lemma 1 *To schedule a collection $\mathcal{J}$ of job sets on P processors of unit speed, the optimal set response time is at least the total span of $\mathcal{J}$, i.e., $H_1^*(\mathcal{J}) \geq l(\mathcal{J})$, and if all job sets are batch released, the optimal set response time also satisfies $H_1^*(\mathcal{J}) \geq \hat{w}(\mathcal{J})$, where $\hat{w}(\mathcal{J})$ is the squashed work of $\mathcal{J}$.*

Proof. It takes at least the span of job J_{ij}, that is $l(J_{ij})$ time, to complete the job on unit-speed processors. According to the definition of set span (Section 2.1), completing job set $\mathcal{J}_i$ then takes at least $l(\mathcal{J}_i)$ time after its release. Thus, the optimal set response time of any job set collection $\mathcal{J}$ satisfies $H_1^*(\mathcal{J}) \geq \sum_{i=1}^{m} l(\mathcal{J}_i) = l(\mathcal{J})$, regardless of the release times of its job sets.

When all job sets in $\mathcal{J}$ are batch released, completing any k sets of jobs, where $1 \leq k \leq m$, takes at least $\frac{1}{P}\sum_{i=m-k+1}^{m} w(\mathcal{J}_{\pi(i)})$ time on P processors of unit-speed. This is because no other schedule can produce a better completion time than executing k job sets that have the least amount of work without wasting any processor resources. The optimal set response time therefore satisfies $H_1^*(\mathcal{J}) \geq \frac{1}{P}\sum_{k=1}^{m}\sum_{i=m-k+1}^{m} w(\mathcal{J}_{\pi(i)}) = \frac{1}{P}\sum_{i=1}^{m} i \cdot w(\mathcal{J}_{\pi(i)}) = \hat{w}(\mathcal{J})$. □

3 The EQUI∘AGDEQ Algorithm

In this section, we introduce a fair and efficient online adaptive scheduling algorithm, called EQUI∘AGDEQ. Specifically, EQUI∘AGDEQ uses the simple algorithm EQUI [29] to achieve fairness among the competing job sets, and it uses the feedback-driven algorithm AGDEQ [11] to achieve efficient execution of the jobs within each job set.

3.1 EQUI Algorithm

The EQUI (Equi-partitioning) algorithm [29] simply divides the total number of processors evenly among all active job sets at any time. Suppose that at time t there are m_t active job sets. Then each active job set $\mathcal{J}_i$ receives an allocation $a(\mathcal{J}_i, t) = P/m_t$ of processors. Hence, EQUI only reallocates the processors whenever a new job set is released or an existing job set is completed. Although simple, this algorithm is able to ensure the basic notion of fairness among the competing job sets by giving each of them the same share of processor resource.

3.2 AGDEQ Algorithm

Within each job set, the efficiency of the allocated processors are guaranteed by consciously exploiting the jobs' execution history and periodically adjusting the processor allocations among them. This is realized by the feedback-driven algorithm AGDEQ [11], which works based on the interaction between the A-GREEDY [1] scheduler and the DEQ [15] allocator after each *scheduling quantum*. Specifically, A-GREEDY collects the execution statistics of a job in each quantum, based on which it calculates the job's *processor desire*, that is, how many processors the job requires, for the next quantum. The DEQ allocator then decides a *processor allocation* for each job in the next quantum according to the processor desires of all the jobs. As this process only involves the execution history of the jobs but not their future characteristics, AGDEQ is indeed a *non-clairvoyant* algorithm [17, 11]. In the following, we will describe the desire calculation strategy of A-GREEDY and the processor allocation policy of DEQ, separately.

A-GREEDY Scheduler

The A-GREEDY scheduler [1] calculates the processor desires of a job using a simple *multiplicative-increase multiplicative-decrease* strategy. For each job J_{ij} in a scheduling quantum q, let $d(J_{ij}, q)$ and $a(J_{ij}, q)$ denote its processor desire and processor allocation, respectively. We say that job J_{ij} is *satisfied* in quantum q if its processor allocation is at least its processor desire, i.e., $a(J_{ij}, q) \geq d(J_{ij}, q)$. Otherwise, the job is said to be *deprived* if $a(J_{ij}, q) < d(J_{ij}, q)$. Let t_q denote the time when quantum q starts and let L denote the duration of the quantum. A-GREEDY collects the amount of work $w(J_{ij}, q)$ completed for job J_{ij} in quantum q, i.e., $w(J_{ij}, q) = \int_{t_q}^{t_q+L} s \cdot \Gamma_{ij}^{k_t}(t)dt$, where k_t is the phase job J_{ij} is executing at time t and s is the processor speed. Given that the total allocated processor cycles for job J_{ij} in quantum q is $a(J_{ij}, q)sL$, job J_{ij} is said to be *efficient* if its completed work is at least δ fraction of the total allocated processor cycles, i.e., $w(J_{ij}, q) \geq \delta \cdot a(J_{ij}, q)sL$, where $\delta \leq 1$ is called the *utilization parameter*. Otherwise, the job is said to be *inefficient* if $w(J_{ij}, q) < \delta \cdot a(J_{ij}, q)sL$. The processor desire $d(J_{ij}, q+1)$ of job J_{ij} in the next quantum $q+1$

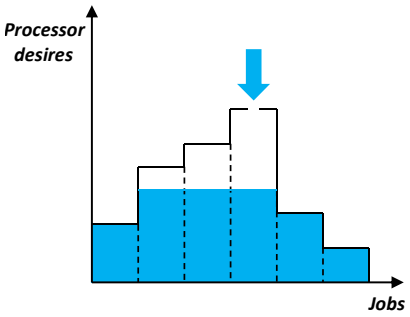

Figure 1: The DEQ allocator viewed as a water-filling algorithm.

is then calculated based on whether the job is satisfied or deprived and whether it is efficient or inefficient in quantum q as shown below:

$$d(J_{ij}, q+1) = \begin{cases} d(J_{ij}, q) \cdot \rho & \text{if } J_{ij} \text{ is efficient and satisfied in } q, \\ d(J_{ij}, q)/\rho & \text{if } J_{ij} \text{ is inefficient in } q, \\ d(J_{ij}, q) & \text{if } J_{ij} \text{ is efficient and deprived in } q, \end{cases}$$

where $\rho > 1$ is called the *responsiveness parameter*.

The rationale behind A-Greedy's desire calculation strategy is as follows [1]: If the allocated processors in a quantum are not utilized efficiently, then the parallelism of the job may not be as high. Therefore, the processor desire will be reduced by a factor of ρ in the next quantum. If the allocated processors are utilized efficiently and the processor desire is satisfied, then the parallelism of the job could be even higher. To exploit any potential parallelism, the processor desire will be increased by a factor of ρ in the next quantum. Lastly, if the allocated processors are utilized efficiently but the desire is deprived, then it is not known whether the processors could have been efficiently utilized had the desire been satisfied. Therefore, the processor desire is unchanged in the next quantum. The initial desire of the job when it first enters the system is simply set as 1 to start with.

Deq Allocator

The Deq (Dynamic Equi-partitioning) allocator [15] distributes the processors among the active jobs within a job set based on the processor desires of all jobs in the set. Similarly to Equi, Deq attempts to ensure fairness among the active jobs by giving each of them an equal share of processor resource. However, Deq is more efficient in the sense that it never allocates more processors to a job than what the job desires, otherwise the extra processors will likely to be wasted in the following quantum assuming that the processor desires of the jobs can reasonably reflect the jobs' actual parallelism. The surplus processors will instead be shared among the other jobs with higher desires. Informally, Deq can be viewed as a water-filling algorithm as shown in Figure 1, where the height of each vertical bar represents the processor desire of the corresponding job and the total amount of water represents the total number of processors possessed by the job set. After pouring the water into the bars, the level of filled water in each bar will turn out to be the number of processors allocated to the respective job. As we can see, Deq tends to satisfy those jobs with low processor desires while the jobs with high desires are likely to be deprived and share an equal processor allocation.

Algorithm 1 presents the pseudocode of the Deq allocator. First, the algorithm initializes the processor allocation $a(J_{ij}, q) = 0$ for each active job $J_{ij} \in \mathcal{J}_i$ (Lines 1-3). Suppose a job set $\mathcal{J}_i$ receives $a(\mathcal{J}_i, q)$ processors from Equi in quantum q. It finds the minimum processor desire $d_{\min}$ among all active jobs in $\mathcal{J}_i$ (Line 5). If the intended equal processor share $a(\mathcal{J}_i, q)/|\mathcal{J}_i|$ does not exceed the minimum desire, the remaining processors will be equally divided among the active jobs (Lines 6-10). Otherwise, the algorithm adds $d_{\min}$ processors to the allocations of all active jobs and updates their remaining desires accordingly (Lines 11-15). The process is then repeated by excluding the processors already allocated and the jobs already satisfied (Lines 16-18). The algorithm terminates when all active jobs in $\mathcal{J}_i$ are satisfied or the initial $a(\mathcal{J}_i, q)$ processors are all distributed to the active jobs (Line 4).

Algorithm 1 DEQ Allocator

Input: Processor allocation $a(\mathcal{J}_i, q)$ of job set $\mathcal{J}_i$, and processor desire $d(J_{ij}, q)$ of each job $J_{ij} \in \mathcal{J}_i$ in quantum q.

Output: Processor allocation $a(J_{ij}, q)$ for each job $J_{ij} \in \mathcal{J}_i$ in quantum q.

```
1: for each J_ij ∈ J_i do
2:    a(J_ij, q) = 0
3: end for
4: while J_i ≠ ∅ and a(J_i, q) > 0 do
5:    d_min = min_{J_ij ∈ J_i} d(J_ij, q)
6:    if d_min ≥ a(J_i, q)/|J_i| then
7:       for each J_ij ∈ J_i do
8:          a(J_ij, q) = a(J_ij, q) + a(J_i, q)/|J_i|
9:       end for
10:      a(J_i, q) = 0
11:   else
12:      for each J_ij ∈ J_i do
13:         a(J_ij, q) = a(J_ij, q) + d_min
14:         d(J_ij, q) = d(J_ij, q) − d_min
15:      end for
16:      a(J_i, q) = a(J_i, q) − d_min · |J_i|
17:      J_i^B = {J_ij ∈ J_i | d(J_ij, q) = 0}
18:      J_i = J_i \ J_i^B
19:   end if
20: end while
```

3.3 Simplifying Assumption

To ease analysis, we assume that any job set is released or completed at the boundary of two consecutive scheduling quanta. Since EQUI is not quantum-based and only reallocates the processors based on the release and the completion of job sets, the assumption ensures that the scheduling decisions made by EQUI at the job-set level and the decisions by AGDEQ at the job level are well synchronized. This assumption can be easily justified as most computation-intensive workloads take much longer to execute compared to any realistic quantum size, which is normally in the order of milliseconds. Hence, a couple of extra quanta on the overall execution time of a job set can be practically ignored.

4 Performance Analysis of EQUI∘AGDEQ

4.1 Preliminaries

To analyze the performance of EQUI∘AGDEQ, we need to define some preliminary concepts and notations. First of all, we extend the notions of "satisfied" and "deprived" from quantum to time as follows: a job is said to be satisfied (or deprived) at time t if t is within a satisfied (or deprived) quantum for the job. In addition, we extend these notions from an individual job to a job set as follows: a job set $\mathcal{J}_i$ is said to be *satisfied* at time t if *all* jobs in $\mathcal{J}_i$ are satisfied at t; otherwise, $\mathcal{J}_i$ is said to be *deprived* if it contains *at least one* deprived job at t. Let $\mathcal{J}_i(t)$ denote job set $\mathcal{J}_i$ at time t, and let $\mathcal{J}(t)$ denote the set of all active job sets at t. Moreover, let $\mathcal{J}_A(t)$ and $\mathcal{J}_B(t)$ denote the set of deprived job sets and the set of satisfied job sets in $\mathcal{J}(t)$, respectively. Throughout the execution of job set $\mathcal{J}_i$, we define $a_A(\mathcal{J}_i)$ to be the amount of processor allocation $\mathcal{J}_i$ receives when it is deprived, or its *deprived processor allocation*, i.e., $a_A(\mathcal{J}_i) = \int_0^\infty a(\mathcal{J}_i, t)s \cdot [\mathcal{J}_i(t) \in \mathcal{J}_A(t)]dt$, and define $t_B(\mathcal{J}_i)$ to be the amount of processor time for $\mathcal{J}_i$ when it is satisfied, or its *satisfied processor time*, i.e., $t_B(\mathcal{J}_i) = \int_0^\infty s \cdot [\mathcal{J}_i(t) \in \mathcal{J}_B(t)]dt$, where s is the speed of the processors used by EQUI∘AGDEQ, and $[x]$ is 1 if proposition x is true and 0 otherwise. To simplify notations, let $m_t^A = |\mathcal{J}_A(t)|$ and $m_t^B = |\mathcal{J}_B(t)|$ denote the number of deprived job sets and the number of satisfied job sets at time t, respectively. Since an active job set is either satisfied or deprived, we have $m_t^A + m_t^B = m_t$, where $m_t = |\mathcal{J}(t)|$ is the total number of active job sets at t.

We now introduce the concepts of *squashed deprived processor allocation* $\hat{a}_A(\mathcal{J})$ and *total satisfied processor*

time $t_B(\mathcal{J})$ for the entire job set collection $\mathcal{J}$ as follows:

$$\hat{a}_A(\mathcal{J}) = \frac{1}{P}\sum_{i=1}^{m} i \cdot a_A(\mathcal{J}_{\gamma(i)}), \quad (1)$$

$$t_B(\mathcal{J}) = \sum_{i=1}^{m} t_B(\mathcal{J}_i), \quad (2)$$

where $\gamma(\cdot)$ denotes a permutation of the job sets sorted in non-increasing order of deprived processor allocation, i.e., $a_A(\mathcal{J}_{\gamma(1)}) \geq a_A(\mathcal{J}_{\gamma(2)}) \geq \cdots \geq a_A(\mathcal{J}_{\gamma(m)})$. It is not difficult to see that $\gamma(\cdot)$, among all permutations of the job sets, gives the minimum value for the squashed formulation, i.e., $\sum_{i=1}^{m} i \cdot a_A(\mathcal{J}_{\gamma(i)}) \leq \sum_{i=1}^{m} i \cdot a_A(\mathcal{J}_{\pi(i)})$ for any permutation $\pi(\cdot)$ of the job sets. The following lemma derives the upper bounds for the squashed deprived processor allocation and the total satisfied processor time in terms of the squashed work and the total span of a job set collection (defined in Section 2.3), respectively. These bounds will be used later in the analysis.

Lemma 2 *Suppose that* EQUI∘AGDEQ *schedules a collection* $\mathcal{J}$ *of* m *job sets on* P *processors of speed* s*, where* $s > 0$*. Then the squashed deprived processor allocation* $\hat{a}_A(\mathcal{J})$ *and the total satisfied processor time* $t_B(\mathcal{J})$ *for the job set collection* $\mathcal{J}$ *satisfy*

$$\hat{a}_A(\mathcal{J}) \leq \frac{1+\rho}{\delta} \cdot \hat{w}(\mathcal{J}), \quad (3)$$

$$t_B(\mathcal{J}) \leq \frac{2}{1-\delta} \cdot l(\mathcal{J}) + msL(\log_\rho P + 1), \quad (4)$$

where $\hat{w}(\mathcal{J})$ *and* $l(\mathcal{J})$ *denote the squashed work and the total span of* $\mathcal{J}$*,* δ *and* ρ *denote* A-GREEDY*'s utilization and responsiveness parameters, and* L *is the quantum length.*

Proof. For any job J_{ij} scheduled by AGDEQ, we define $a(J_{ij})$ to be its total processor allocation, i.e., $a(J_{ij}) = \int_0^\infty a(J_{ij}, t)sdt$, and define $t_B(J_{ij})$ to be its satisfied processor time, i.e., $t_B(J_{ij}) = \int_0^\infty s \cdot [J_{ij}(t) \in \mathcal{J}_i^B(t)]dt$, where $\mathcal{J}_i^B(t)$ denotes the set of satisfied jobs in job set $\mathcal{J}_i$ at time t. The results in [1, 11] show that on P processors of speed s, where $s > 0$, the A-GREEDY scheduler achieves $a(J_{ij}) \leq \frac{1+\rho}{\delta} \cdot w(J_{ij})$ and $t_B(J_{ij}) \leq \frac{2}{1-\delta} \cdot l(J_{ij}) + sL(\log_\rho P + 1)$, if the job never receives more processors than what it desires, which is obviously satisfied by the DEQ allocator.

According to definition, the satisfied processor time for job set $\mathcal{J}_i$ is then given by $t_B(\mathcal{J}_i) \leq \max_{j=1\cdots n_i} t_B(J_{ij}) \leq \frac{2}{1-\delta} \max_{j=1\cdots n_i} l(J_{ij}) + sL(\log_\rho P + 1) = \frac{2}{1-\delta} \cdot l(\mathcal{J}_i) + sL(\log_\rho P + 1)$. When job set $\mathcal{J}_i$ is deprived, which means that at least one job within $\mathcal{J}_i$ is deprived, according to the DEQ allocator, all the processors allocated to $\mathcal{J}_i$ must have been distributed to its jobs. Hence, the deprived processor allocation for $\mathcal{J}_i$ satisfies $a_A(\mathcal{J}_i) \leq \sum_{j=1}^{n_i} a(J_{ij}) \leq \frac{1+\rho}{\delta} \sum_{j=1}^{n_i} w(J_{ij}) = \frac{1+\rho}{\delta} \cdot w(\mathcal{J}_i)$. The squashed deprived processor allocation for the entire job set collection $\mathcal{J}$ is then given by $\hat{a}_A(\mathcal{J}) = \frac{1}{P}\sum_{i=1}^{m} i \cdot a_A(\mathcal{J}_{\gamma(i)}) \leq \frac{1}{P}\sum_{i=1}^{m} i \cdot a_A(\mathcal{J}_{\pi(i)}) \leq \frac{1}{P}\sum_{i=1}^{m} i \cdot \frac{1+\rho}{\delta} \cdot w(\mathcal{J}_{\pi(i)}) = \frac{1+\rho}{\delta} \cdot \hat{w}(\mathcal{J})$. The total satisfied processor time for $\mathcal{J}$ can be obtained by summing up the satisfied processor time over all job sets. □

Finally, we introduce the notions of t-prefix and t-suffix. For EQUI∘AGDEQ, we define the *t-prefix* $\mathcal{J}_i(\overleftarrow{t})$ for job set $\mathcal{J}_i$ to be the portion of the job set executed before and at time t, and define its *t-suffix* $\mathcal{J}_i(\overrightarrow{t})$ to be the portion executed after time t. We then extend the notions of t-prefix and t-suffix to the job set collection as follows: the t-prefix of job set collection $\mathcal{J}$ is defined to be $\mathcal{J}(\overleftarrow{t}) = \{\mathcal{J}_i(\overleftarrow{t}) : \mathcal{J}_i \in \mathcal{J} \text{ and } r_i \leq t\}$ and the t-suffix of $\mathcal{J}$ is $\mathcal{J}(\overrightarrow{t}) = \{\mathcal{J}_i(\overrightarrow{t}) : \mathcal{J}_i \in \mathcal{J} \text{ and } r_i \leq t\}$. Note that both t-prefix and t-suffix are defined for job sets that have been released by time t, i.e., $r_i \leq t$. Similarly, we define $\mathcal{J}^*(\overleftarrow{t})$ and $\mathcal{J}^*(\overrightarrow{t})$ to be the t-prefix and the t-suffix for the job set collection $\mathcal{J}$ executed by the optimal offline scheduler.

4.2 Analysis for Batched Job Sets

We first analyze the set response time of the EQUI∘AGDEQ algorithm when all job sets are batch released. The analysis relies on the *local competitiveness argument* [18], which bounds the performance of an online algorithm at any time in terms of the optimal offline scheduler, or in this case its two lower bounds presented in Section 2.3.

For any job set collection $\mathcal{J}$, we focus on its t-prefix $\mathcal{J}(\overleftarrow{t})$, which according to definition always contains m sets of jobs for any $t > 0$. Recall that the squashed deprived processor allocation for $\mathcal{J}(\overleftarrow{t})$ is given by $\hat{a}_A(\mathcal{J}(\overleftarrow{t})) = \frac{1}{P}\sum_{i=1}^{m} i \cdot a_A(\mathcal{J}_{\gamma(i)}(\overleftarrow{t}))$, where $\gamma(\cdot)$ denotes a permutation of the job sets in $\mathcal{J}(\overleftarrow{t})$ sorted in non-increasing order of deprived processor allocation. At any time t, let $m_t(z)$ denote the number of job sets in $\mathcal{J}(\overleftarrow{t})$ whose deprived processor allocation is at least z under EQUI∘AGDEQ, i.e., $m_t(z) = \sum_{\mathcal{J}_i(\overleftarrow{t}) \in \mathcal{J}(\overleftarrow{t})} [a_A(\mathcal{J}_i(\overleftarrow{t})) \geq$

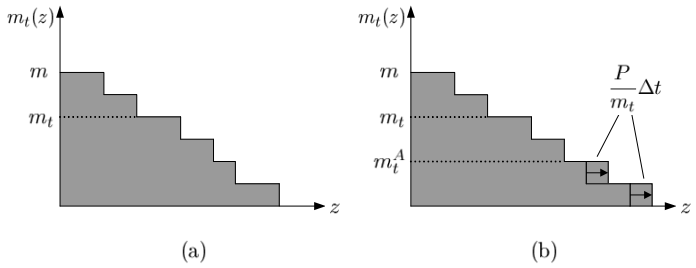

Figure 2: (a) An example of $m_t(z)$ at a particular time t. (b) The changes of $m_t(z)$ in an infinitesimal interval Δt in the worst case.

$z]$. Apparently, $m_t(z)$ is a staircase-like decreasing function of z, and Figure 2(a) shows an example of $m_t(z)$ at a particular time t. From the definition of squashed deprived processor allocation, an alternative expression for $\hat{a}_A(\mathcal{J}(\overleftarrow{t}))$ is given by

$$\hat{a}_A(\mathcal{J}(\overleftarrow{t})) = \frac{1}{P}\int_0^\infty \left(\sum_{i=1}^{m_t(z)} i\right) dz, \tag{5}$$

which is a more convenient representation of $\hat{a}_A(\mathcal{J}(\overleftarrow{t}))$ for analyzing batched job sets. Note that the expression presented in Eq. (5) gives a much simpler perspective for the squashed deprived processor allocation than the one conceived in [11], hence it can be applied to simplify the batched response time analysis therein. For our analysis on set response time, we give both EQUI∘AGDEQ and the optimal scheduler P processors of unit speed. The local performance of EQUI∘AGDEQ is shown in the following lemma.

Lemma 3 *Suppose that* EQUI∘AGDEQ *schedules a collection $\mathcal{J}$ of batched job sets on P processors of unit speed. Then the execution of the job sets satisfies the following*

- *Running condition:* $m_t \le 2\left(\frac{d\hat{a}_A(\mathcal{J}(t))}{dt} + m_t^B\right)$,

where $\frac{d\hat{a}_A(\mathcal{J}(t))}{dt} = \frac{\hat{a}_A(\mathcal{J}(\overleftarrow{t+\Delta t})) - \hat{a}_A(\mathcal{J}(\overleftarrow{t}))}{\Delta t}$ *denotes the rate of change for the squashed deprived processor allocation in an infinitesimal interval Δt during which no job set completes.*

Proof. According to the EQUI algorithm, each of the m_t active job sets gets P/m_t processors at time t. In the worst case, the m_t^A deprived job sets have the most deprived processor allocation so far among the m_t active job sets. As a result, in an infinitesimal interval Δt during which no job set completes, each of the bottom m_t^A horizontal stripes from $m_t(z)$ grows by $\frac{P}{m_t}\Delta t$, as shown in Figure 2(b). The rate of change for the squashed deprived processor allocation can then be bounded by

$$\begin{aligned}\frac{d\hat{a}_A(\mathcal{J}(t))}{dt} &= \frac{1}{P\Delta t}\int_0^\infty \left[\left(\sum_{i=1}^{m_{t+\Delta t}(z)} i\right) - \left(\sum_{i=1}^{m_t(z)} i\right)\right] dz \\ &\ge \frac{1}{P\Delta t}\cdot\frac{m_t^A(m_t^A+1)}{2}\cdot\frac{P}{m_t}\Delta t \ge \frac{x_t^2}{2}m_t,\end{aligned} \tag{6}$$

where $x_t = m_t^A/m_t$, and obviously $0 \le x_t \le 1$. Since a job set is either satisfied or deprived, we have $m_t^B = (1-x_t)m_t$. It can be easily verified that the running condition holds for all values of x_t by substituting Inequality (6) into it. □

We can now combine the results of Lemmas 2 and 3 to get the set response time of EQUI∘AGDEQ in the batched scenario.

Theorem 1 *Suppose that* EQUI∘AGDEQ *schedules a collection $\mathcal{J}$ of m batched job sets on P processors of unit speed. Then its set response time satisfies*

$$H_1(\mathcal{J}) \le \frac{2(1+\rho+\delta-\rho\delta)}{\delta(1-\delta)}H_1^*(\mathcal{J}) + 2mL(\log_\rho P + 1),$$

where $H_1^*(\mathcal{J})$ denotes the set response time of $\mathcal{J}$ under the optimal offline scheduler on unit-speed processors, δ and ρ denote A-GREEDY*'s utilization and responsiveness parameters, and L is the quantum length.*

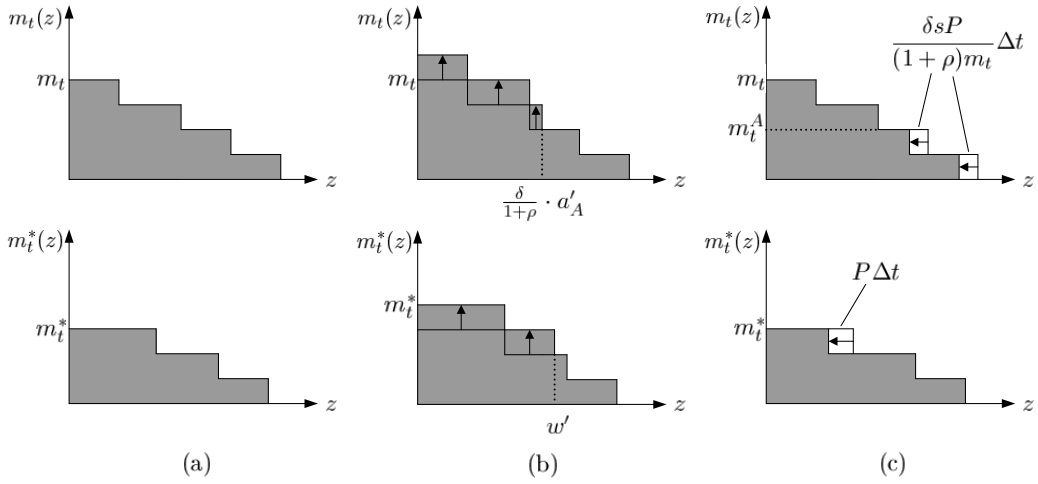

Figure 3: (a) An example of $m_t(z)$ and $m_t^*(z)$ at a particular time t. (b) The changes of $m_t(z)$ and $m_t^*(z)$ after a new job set with work w' and deprived processor allocation a'_A arrives. (c) The changes of $m_t(z)$ and $m_t^*(z)$ in an infinitesimal interval Δt in the worst case.

Proof. As mentioned in Section 2.2, the set response time of a job set collection $\mathcal{J}$ scheduled by EQUI $\circ$ AGDEQ can be expressed as $H_1(\mathcal{J}) = \int_0^\infty m_t dt$. Similarly, the total satisfied processor time of $\mathcal{J}$ under EQUI$\circ$AGDEQ is given by $t_B(\mathcal{J}) = \int_0^\infty m_t^B dt$. Integrating the running condition in Lemma 3, we have $H_1(\mathcal{J}) \le 2\left(\hat{a}_A(\mathcal{J}) + t_B(\mathcal{J})\right)$. Substituting the bounds of $\hat{a}_A(\mathcal{J})$ and $t_B(\mathcal{J})$ from Lemma 2 into the above inequality, we get $H_1(\mathcal{J}) \le 2\left(\frac{1+\rho}{\delta} \cdot \hat{w}(\mathcal{J}) + \frac{2}{1-\delta} \cdot l(\mathcal{J}) + mL(\log_\rho P + 1)\right)$. Based on Lemma 1, both squashed work $\hat{w}(\mathcal{J})$ and total span $l(\mathcal{J})$ are lower bounds on the set response time of job set collection $\mathcal{J}$. The performance of EQUI$\circ$AGDEQ thus satisfies $H_1(\mathcal{J}) \le 2\left(\frac{1+\rho}{\delta} \cdot H_1^*(\mathcal{J}) + \frac{2}{1-\delta} \cdot H_1^*(\mathcal{J}) + mL(\log_\rho P + 1)\right) = \frac{2(1+\rho+\delta-\rho\delta)}{\delta(1-\delta)} H_1^*(\mathcal{J}) + 2mL(\log_\rho P + 1)$. □

4.3 Analysis for Arbitrary Released Job Sets

We now analyze the set response time of EQUI$\circ$AGDEQ when the job sets can have arbitrary release time. Note that the squashed work is no longer a lower bound for the set response time in this scenario. Hence, the analysis uses the *amortized local competitiveness argument* [18], which bounds the performance of an online algorithm at any time in terms of the optimal offline scheduler with the help of a potential function.

We adopt the potential function in [24] designed for the response time analysis of AGDEQ. However, compared to the previous potential function, where only processor allocations of deprived jobs are considered, the one used here needs to incorporate the whole job set collection instead of a single job set. Hence, for each deprived job set at any time, the potential function considers its entire processor allocation, including those from satisfied jobs. In particular, we focus on the t-suffix $\mathcal{J}(\overrightarrow{t})$ of job set collection $\mathcal{J}$, and define $m_t(z)$ to be the number of job sets in $\mathcal{J}(\overrightarrow{t})$ whose deprived processor allocation is at least $\frac{1+\rho}{\delta} \cdot z$ at time t under EQUI$\circ$AGDEQ, i.e., $m_t(z) = \sum_{\mathcal{J}_i(\overrightarrow{t}) \in \mathcal{J}(\overrightarrow{t})} [a_A(\mathcal{J}_i(\overrightarrow{t})) \ge \frac{1+\rho}{\delta} \cdot z]$. Moreover, let $m_t^*(z)$ denote the number of job sets in $\mathcal{J}^*(\overrightarrow{t})$ whose work is at least z under the optimal, i.e., $m_t^*(z) = \sum_{\mathcal{J}_i^*(\overrightarrow{t}) \in \mathcal{J}^*(\overrightarrow{t})} [w(\mathcal{J}_i^*(\overrightarrow{t})) \ge z]$. Hence, both $m_t(z)$ and $m_t^*(z)$ are staircase-like decreasing functions of z, and Figure 3(a) shows an example of $m_t(z)$ and $m_t^*(z)$ at a particular time t. We give the EQUI$\circ$AGDEQ algorithm P processors of speed s, where $s = \frac{2(1+\rho)}{\delta} + \epsilon$ for any $\epsilon > 0$, while the optimal scheduler uses unit-speed processors. The potential function is defined as

$$\Phi(t) = \eta \int_0^\infty \left[\left(\sum_{i=1}^{m_t(z)} i \right) - m_t(z) m_t^*(z) \right] dz, \tag{7}$$

where $\eta = \frac{2(1+\rho)}{\delta\epsilon P}$. We now prove the amortized local performance of EQUI$\circ$AGDEQ in the following lemma.

Lemma 4 *Suppose that* EQUI$\circ$AGDEQ *schedules a collection $\mathcal{J}$ of job sets on P processors of speed s, where $s = \frac{2(1+\rho)}{\delta} + \epsilon$ for any $\epsilon > 0$. Then with the potential function defined in Eq. (7), the execution of the job sets satisfies the following*

- *Boundary condition:* $\Phi(0) = 0$ *and* $\Phi(\infty) = 0$*;*
- *Arrival condition:* $\Phi(t)$ *does not increase when a new job set arrives;*
- *Running condition:* $m_t + \frac{d\Phi(t)}{dt} \le \frac{2s}{\epsilon}\left(m_t^* + m_t^B\right)$*,*

where $\frac{d\Phi(t)}{dt} = \frac{\Phi(t+\Delta t)-\Phi(t)}{\Delta t}$ *denotes the rate of change for the potential function in an infinitesimal interval* Δt *during which no job set arrives or completes under either* EQUI∘AGDEQ *or the optimal offline scheduler.*

Proof. We check each condition in the following.

- *Boundary condition*: at time 0, no job set exists in the system. Therefore, we have $\Phi(0) = 0$. At time ∞, all job sets have completed execution, so again we have $\Phi(\infty) = 0$.

- *Arrival condition*: suppose that a new job set with work w' arrives at time t. Let t^- and t^+ denote the time instances right before and after the job set arrives. Hence, we have $m_{t^+}^*(z) = m_{t^-}^*(z) + 1$ for $z \le w'$ and $m_{t^+}^*(z) = m_{t^-}^*(z)$ for $z > w'$. Similarly, $m_{t^+}(z) = m_{t^-}(z) + 1$ for $z \le \frac{\delta}{1+\rho} \cdot a_A'$ and $m_{t^+}(z) = m_{t^-}(z)$ for $z > \frac{\delta}{1+\rho} \cdot a_A'$, where a_A' is the deprived processor allocation for the job set. Figure 3(b) illustrates the changes of $m_t(z)$ and $m_t^*(z)$ in this case. Note that $\frac{\delta}{1+\rho} \cdot a_A' \le w'$ from the proof of Lemma 2. For convenience, we define $\phi_t(z) = \left(\sum_{i=1}^{m_t(z)} i\right) - m_t(z)m_t^*(z)$. It is obvious that for $z > w'$, we have $\phi_{t^+}(z) = \phi_{t^-}(z)$. For $z \le w'$, we consider two cases.

Case 1: for $z \le \frac{\delta}{1+\rho} \cdot a_A'$, we have $\phi_{t^+}(z) - \phi_{t^-}(z) = \left(\sum_{i=1}^{m_{t^-}(z)+1} i\right) - \left(m_{t^-}(z)+1\right)\left(m_{t^-}^*(z)+1\right) - \left(\sum_{i=1}^{m_{t^-}(z)} i\right) + m_{t^-}(z)m_{t^-}^*(z) = -m_{t^-}^*(z) \le 0$.

Case 2: for $\frac{\delta}{1+\rho} \cdot a_A' \le z \le w'$, we have $\phi_{t^+}(z) - \phi_{t^-}(z) = \left(\sum_{i=1}^{m_{t^-}(z)} i\right) - m_{t^-}(z)\left(m_{t^-}^*(z)+1\right) - \left(\sum_{i=1}^{m_{t^-}(z)} i\right) + m_{t^-}(z)m_{t^-}^*(z) = -m_{t^-}(z) \le 0$.

Thus, $\Phi(t^+) = \eta \int_0^\infty \phi_{t^+}(z)dz \le \eta \int_0^\infty \phi_{t^-}(z)dz = \Phi(t^-)$.

- *Running condition*: Each of the m_t active job sets gets P/m_t processors according to EQUI. In the worst case, the m_t^A deprived job sets have the most remaining deprived processor allocation, while the optimal uses all P processors. Hence, in an infinitesimal interval Δt where no job set arrives or completes, each of the bottom m_t^A horizontal stripes from $m_t(z)$ shrinks by $\frac{\delta s P}{(1+\rho)m_t}\Delta t$, and the horizontal stripes from $m_t^*(z)$ shrinks by $P\Delta t$ totally, as shown in Figure 3(c). The rate of change for the potential function can then be bounded by

$$\begin{aligned}
\frac{d\Phi(t)}{dt} &= \frac{\eta}{\Delta t}\int_0^\infty \left[\left(\sum_{i=1}^{m_{t+\Delta t}(z)} i\right) - m_{t+\Delta t}(z)m_{t+\Delta t}^*(z) - \left(\sum_{i=1}^{m_t(z)} i\right) + m_t(z)m_t^*(z)\right] dz \\
&\le \frac{2(1+\rho)}{\delta\epsilon P\Delta t}\left(-\frac{m_t^A(m_t^A+1)}{2}\cdot\frac{\delta s P}{(1+\rho)m_t}\Delta t + m_t P\Delta t + m_t^*\frac{\delta s P}{(1+\rho)m_t}\Delta t \cdot m_t^A\right) \\
&\le \frac{2(1+\rho)}{\delta\epsilon}\left(1 - \frac{\delta s x_t^2}{2(1+\rho)}\right) m_t + \frac{2s}{\epsilon}m_t^*, \qquad (8)
\end{aligned}$$

where $x_t = m_t^A/m_t$, and $0 \le x_t \le 1$. Since a job set is either satisfied or deprived, we have $m_t^B = (1-x_t)m_t$. We can again verify that the running condition holds for all values of x_t by substituting Inequality (8) into it. □

The following theorem gives the set response time of EQUI∘AGDEQ for arbitrary released job sets.

Theorem 2 *Suppose that* EQUI∘AGDEQ *schedules a collection* $\mathcal{J}$ *of* m *job sets on* P *processors of speed* s*, where* $s = \frac{2(1+\rho)}{\delta} + \epsilon$ *for any* $\epsilon > 0$*. Then its set response time satisfies*

$$H_s(\mathcal{J}) \le \left(2 + \frac{4(1+\rho-\rho\delta)}{\delta(1-\delta)\epsilon}\right) H_1^*(\mathcal{J}) + \frac{2msL}{\epsilon}(\log_\rho P + 1),$$

where $H_1^*(\mathcal{J})$ *denotes the set response time of* $\mathcal{J}$ *under the optimal scheduler on unit-speed processors,* δ *and* ρ *denote* A-GREEDY*'s utilization and responsiveness parameters, and* L *is the quantum length.*

Proof. As the set response time of EQUI∘AGDEQ is given by $H_s(\mathcal{J}) = \int_0^\infty m_t dt$, and the set response time of the optimal is $H_1^*(\mathcal{J}) = \int_0^\infty m_t^* dt$, integrating the running condition in Lemma 4 over time and applying the boundary and arrival conditions, we have $H_s(\mathcal{J}) \le \frac{2s}{\epsilon} \cdot H_1^*(\mathcal{J}) + \frac{2}{\epsilon} \cdot t_B(\mathcal{J})$, where $t_B(\mathcal{J}) = \int_0^\infty s \cdot m_t^B dt$ is the total satisfied processor time for $\mathcal{J}$ under EQUI∘AGDEQ. From Lemma 2, the total satisfied processor time for $\mathcal{J}$ is also given by $t_B(\mathcal{J}) \le \frac{2}{1-\delta} \cdot l(\mathcal{J}) + msL(\log_\rho P + 1)$. The set response time of $\mathcal{J}$ scheduled by EQUI∘AGDEQ thus satisfies $H_s(\mathcal{J}) \le \frac{2s}{\epsilon} \cdot H_1^*(\mathcal{J}) + \frac{4}{(1-\delta)\epsilon} \cdot l(\mathcal{J}) + \frac{2msL}{\epsilon}(\log_\rho P + 1)$. Since the total span $l(\mathcal{J})$ is a lower bound for the set response time of $\mathcal{J}$ on unit-speed processors, the theorem is directly implied. □

4.4 Discussions

In the preceding two subsections, we have analyzed the set response time of the EQUI∘AGDEQ algorithm for both batched and arbitrary released job sets. As Theorem 1 shows, when all job sets are batch released, EQUI∘AGDEQ is $O(1)$-competitive, since both ρ and δ can be considered as constants. In particular, when $\delta = 0.5$ and ρ approaches 1, the competitive ratio approaches the minimum value 16. This result improves upon the $O(\frac{\ln n}{\ln \ln n})$-competitiveness of EQUI∘EQUI obtained in [19] for large values of n, where n denotes the maximum number of jobs in a set. For arbitrary released job sets, Theorem 2 shows that EQUI∘AGDEQ is $O(1)$-speed $O(1)$-competitive. When $\delta \approx 0.586$ and ρ approaches 1 in this case, the competitive ratio approaches the minimum value $2 + 23.32/\epsilon$ with $(6.83 + \epsilon)$-speed processors, for any $\epsilon > 0$. This result extends a similar performance bound of the EQUI∘A algorithm obtained in [20] from jobs with specific parallelism structure, namely, a sequential phase followed by a fully-parallelizable phase, to jobs with any parallelism profile.

Note that, for jobs with arbitrary sizes, any non-clairvoyant algorithm has been shown to be $\omega(1)$-competitive in the strong sense with respect to the set response time even when the job sets are batch released [19]. Hence, the competitive ratios of EQUI∘AGDEQ derived in this paper are actually achieved in the asymptotic sense. Assuming that the jobs under consideration are sufficiently large, the optimal set response time will dominate the additive factors shown in the inequalities of Theorems 1 and 2. As a result, the strong competitive ratios will increase by at most an additive constant from the corresponding asymptotic ones. For instance, when job sets are batch released and the optimal set response time is much larger than $2mL(\log_\rho P + 2)$, the performance of EQUI∘AGDEQ as shown in Theorem 1 then becomes $H_1(\mathcal{J}) \le \left(\frac{2(1+\rho+\delta-\rho\delta)}{\delta(1-\delta)} + o(1)\right) H_1^*(\mathcal{J}) = O(1) \cdot H_1^*(\mathcal{J})$.

While both EQUI∘AGDEQ and EQUI∘EQUI use the EQUI algorithm at the job-set level to ensure fairness, the performance improvement of EQUI∘AGDEQ for sufficiently large jobs is essentially because the processors are utilized more efficiently by the AGDEQ algorithm at the job level. On the other hand, EQUI lacks efficiency by obliviously allocating processors to the jobs, which will inevitably incur a large waste of resources when the jobs' parallelism can vary with time. This shows the importance of both fairness among the job sets and efficiency within each job set when scheduling a job set collection for the objective of set response time.

5 A Generalized Analysis Framework for EQUI∘XY

In this section, we extend the performance analysis of the EQUI∘AGDEQ algorithm shown in the preceding section and present a generalized analysis framework for the EQUI∘XY family of algorithms. In particular, EQUI∘XY uses EQUI to allocate processors among the job sets and any feedback-driven algorithm XY to schedule the jobs within each job set. The performance of EQUI∘XY then relies on bounding the efficiency of the XY algorithm. We show the generality of this framework by applying it to two other choices of XY with provable efficiency.

5.1 Generalized Analysis

It was shown in [20] that, for arbitrary released job sets, the EQUI∘A family of algorithms is $O(1)$-speed $O(1)$-competitive with respect to the set response time when all jobs have special parallelism structures, i.e., a sequential phase followed by a fully-parallelizable phase. The result holds independent of the choice of algorithm A, as long as A distributes all processors allocated to a job set to its active jobs at all time, i.e., A never idles its received processors. In contrast, our generalized analysis for the EQUI∘XY algorithm family can be applied to both batched and arbitrary released job sets, and moreover it allows the jobs to have arbitrary parallelism profile. While EQUI ensures fairness of processor allocations among the job sets, the key to the generalized analysis essentially relies on establishing the efficiency of the feedback-driven algorithm XY, which uses the scheduler X to calculate the processor desire for each individual job and the allocator Y to distribute the processors based on the desires of the jobs.

In the generalized analysis, the efficiency of XY is expressed by two conditions, which need to be satisfied by the scheduler X and the allocator Y, respectively. First, for the scheduler X, we need to bound the total processor allocation of any individual job scheduled by it in terms of the job's work, and bound the overall satisfied processor time of the job in terms of the job's span. Specifically, for any job J_{ij} scheduled by X on processors of speed s, where $s > 0$, let $a(J_{ij}) = \int_0^\infty a(J_{ij}, t)sdt$ denote the job's total processor allocation and let $t_B(J_{ij}) = \int_0^\infty s \cdot [J_{ij}(t) \in \mathcal{J}_i^B(t)]dt$ denote its satisfied processor time. The following condition states the performance of scheduler X.

Condition 1 *For any job J_{ij} scheduled by* X *on processors of speed s, where $s > 0$, assuming that the job never*

receives more processors than its desire, the execution of the job should satisfy

$$a(J_{ij}) \leq \alpha \cdot w(J_{ij}), \tag{9}$$
$$t_B(J_{ij}) \leq \beta \cdot l(J_{ij}), \tag{10}$$

where $w(J_{ij})$ and $l(J_{ij})$ denote the work and span of job J_{ij}, respectively.

The coefficients α and β measure the efficiency of the scheduler X, and they should be bounded as minimally as possible in order to achieve the best competitive ratio. For job set $\mathcal{J}_i$, we say that it is *satisfied* at time t if all jobs in $\mathcal{J}_i$ are satisfied. Otherwise, $\mathcal{J}_i$ is said to be *deprived* if it contains at least one deprived job at time t. In order for our analysis to hold, it is also required that the allocator Y satisfies the following condition.

Condition 2 *At any time t, the allocator* Y *should be*

1. Conservative*: Y never allocates more processors to a job than what the job desires.*
2. Non-idle*: When a job set is deprived,* Y *never idles processors, that is, all the processors allocated to the job set must be distributed to the active jobs in the set.*

While Condition 2.1 is required to bound the total processor allocation of a job derived in Inequality (9), Condition 2.2 is necessary to bound the squashed processor allocation to be shown in Theorem 3. Apparently, the DEQ allocator satisfies both conditions.

For any scheduler X satisfying Condition 1 and any allocator Y satisfying Condition 2, we can follow the analysis in Section 4 and show the performance of the EQUI∘XY algorithm. The following theorem gives the generalized results.

Theorem 3 *Suppose that* EQUI∘XY *schedules a collection $\mathcal{J}$ of job sets. If the scheduler* X *satisfies the performance stated in Condition 1 for any individual job and the allocator* Y *satisfies Condition 2, then* EQUI∘XY *is*

- $2(\alpha+\beta)$*-competitive for batched job sets;*
- $(2\alpha+\epsilon)$*-speed* $(2+2(2\alpha+\beta)/\epsilon)$*-competitive for arbitrary released job sets, where* $\epsilon > 0$*,*

with respect to the set response time of the job set collection $\mathcal{J}$.

Proof sketch. As in Section 4.1, we can define the squashed deprived processor allocation $\hat{a}_A(\mathcal{J})$ and the total satisfied processor time $t_B(\mathcal{J})$ for the entire job set collection $\mathcal{J}$. Then based on the derivations of Lemma 2 and the two conditions stated for scheduler X and allocator Y above, they can be shown to satisfy $\hat{a}_A(\mathcal{J}) \leq \alpha \cdot \hat{w}(\mathcal{J})$ and $t_B(\mathcal{J}) \leq \beta \cdot l(\mathcal{J})$, where $\hat{w}(\mathcal{J})$ and $l(\mathcal{J})$ denote the squashed work and the total span of the job set collection $\mathcal{J}$, respectively. The claim can then be proved by following the analysis in Sections 4.2 and 4.3. □

As with the analysis of EQUI∘AGDEQ, if the inequalities in Condition 1 contain any additive constant, the competitive ratios of EQUI∘XY stated in Theorem 3 will hold in the asymptotic sense, under the assumption that the jobs under consideration are sufficiently large.

5.2 Application of the Framework

The preceding subsection outlines an analysis framework, which can be seen to encompass the results of the EQUI∘AGDEQ algorithm with $\alpha = (1+\rho)/\delta$ and $\beta = 2/(1-\delta)$. We now demonstrate the generality of the framework by applying it to two other choices of X with different desire calculation schemes and an alternative allocator Y. First, observe that the analysis framework can be applied to any online algorithm EQUI∘XY, where the scheduler X does not necessarily need to be non-clairvoyant when calculating the processor desires. One particular algorithm (IGCP) we present in this subsection is in fact *IP-clairvoyant* [27], that is, the algorithm knows the instantaneous parallelism (IP) of the job at any time but not the job's future parallelism and remaining work. Not surprisingly, such information does help the algorithm achieve better efficiency and hence stronger performance bounds.

Table 1: Performance of EQUI∘AGDEQ, EQUI∘ACDEQ and EQUI∘IGCP for both batched job sets and arbitrary released job sets, obtained using the generalized analysis framework.

	α	β	Batched job sets	Arbitrary released job sets
EQUI∘AGDEQ	$\frac{1+\rho}{\delta}$	$\frac{2}{1-\delta}$	$\frac{2(1+\rho+\delta-\rho\delta)}{\delta(1-\delta)}$-comp.	$\left(\frac{2(1+\rho)}{\delta}+\epsilon\right)$-speed $\left(2+\frac{4(1+\rho-\rho\delta)}{\delta(1-\delta)\epsilon}\right)$-comp.
EQUI∘ACDEQ	$c+1$	$c+1$	$4(c+1)$-comp.	$(2(c+1)+\epsilon)$-speed $\left(2+\frac{6(c+1)}{\epsilon}\right)$-comp.
EQUI∘IGCP	1	1	4-comp.	$(2+\epsilon)$-speed $(2+6/\epsilon)$-comp.

EQUI∘ACDEQ Algorithm

As with AGDEQ, the ACDEQ algorithm also uses the DEQ allocator to distribute processors among the jobs in a job set, but it uses a more stable desire calculation strategy, namely, the A-CONTROL scheduler [28, 25], to calculate the processor desires. Specifically, A-CONTROL estimates the average parallelism of a job in each scheduling quantum and then directly sets the processor desire of the job in the next quantum to be the current average parallelism. Such strategy makes the processor desire more representative of the job's immediate average processor requirement. Moreover, its feedback over time has been shown to possess desirable transient and steady-state behaviors through formal control-theoretic arguments [25].

To apply the generalized analysis to EQUI∘ACDEQ, the results in [25] show that the total processor allocation $a(J_{ij})$ and the satisfied processor time $t_B(J_{ij})$ for any job J_{ij} scheduled by A-CONTROL can be bounded in terms of the *transition factor* c of the job, and they are given by $a(J_{ij}) \le (c+1)\cdot w(J_{ij})$ and $t_B(J_{ij}) \le (c+1)\cdot l(J_{ij})$. The transition factor captures the job's maximum parallelism change between any two consecutive quanta, and to a certain extent, it reflects the degree of difficulty to schedule the job in a non-clairvoyant manner. For jobs with smooth parallelism variations, c can generally be considered as a constant.

EQUI∘IGCP Algorithm

Unlike the non-clairvoyant algorithms AGDEQ and ACDEQ, the IGCP algorithm is IP-clairvoyant, which means that it is aware of the instantaneous parallelism of the job at any time. In many parallel systems, instantaneous parallelism can be obtained by simply counting the number of available tasks in the queue or the number of ready threads in the pool, which is information practically accessible by the scheduler. In this case, IGCP uses the I-GREEDY scheduler [12] to directly report the instantaneous parallelism of a job at any time as its processor desire, and this has been shown to provide the strong result of $\alpha = \beta = 1$, that is, the total processor allocation $a(J_{ij})$ and the satisfied processor time $t_B(J_{ij})$ for job J_{ij} can be bounded as $a(J_{ij}) \le w(J_{ij})$ and $t_B(J_{ij}) \le l(J_{ij})$ [12].

IGCP then applies the CP (Capped Proportional) allocator to distribute the $a(\mathcal{J}_i, t)$ processors received by the job set $\mathcal{J}_i$ at any time t to its active jobs proportionally based on their desires, or the instantaneous parallelism in this case. However, when all jobs in the set can be satisfied, CP ensures that the allocations are capped at the corresponding desires of the jobs. The following gives the detailed allocation policy:

$$a(J_{ij}, t) = \begin{cases} \frac{d(J_{ij},\, t)}{\sum_{J_{ik}\in\mathcal{J}_i} d(J_{ik},\, t)} \cdot a(\mathcal{J}_i, t) & \text{if } \sum_{J_{ik}\in\mathcal{J}_i} d(J_{ik}, t) > a(\mathcal{J}_i, t), \\ d(J_{ij}, t) & \text{if } \sum_{J_{ik}\in\mathcal{J}_i} d(J_{ik}, t) \le a(\mathcal{J}_i, t), \end{cases}$$

where $a(J_{ij}, t)$ and $d(J_{ij}, t)$ denote the processor allocation and processor desire of job J_{ij} at time t, respectively. Apparently, CP also satisfies both conditions stated in Condition 2. What distinguishes it from DEQ is that all jobs will be deprived if their total desire is more than the number of available processors, whereas in DEQ some jobs with small desire may still be satisfied.

Table 1 summarizes the performance of the three algorithms shown in this paper using the generalized analysis framework. Note that as both allocators DEQ and CP satisfy Condition 2, the three schedulers (A-GREEDY, A-CONTROL and I-GREEDY) can in fact be coupled with any of the two allocators to achieve essentially the same performance as stated in the table.

6 Multi-level Hierarchical Scheduling and a Fair and Efficient Solution

In this section, we extend the two-level scheduling model to a multi-level hierarchical scheduling problem. We propose a fair and efficient solution for allocating resources in this hierarchical model, and show that it achieves the same performance bounds as in the two-level case.

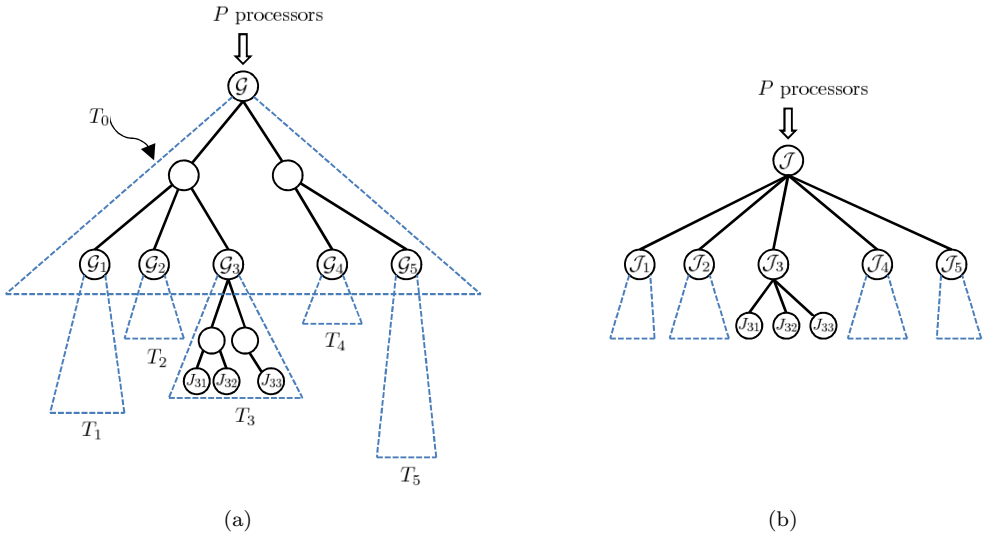

Figure 4: (a) An example of the multi-level hierarchical scheduling problem with five job groups. (b) The reduced two-level scheduling instance using the FAIR∘EFFICIENT solution.

6.1 Multi-level Hierarchical Scheduling

In the multi-level hierarchical scheduling model, we have a collection $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_m\}$ of m job groups, where each group $\mathcal{G}_i = \{J_{i1}, J_{i2}, \cdots, J_{in_i}\}$ contains n_i jobs. Unlike the two-level model with simple flat structures, the job groups here form the leaf nodes of an arbitrary tree structure T_0. A total number P of processors need to be distributed from the root of the tree to the job groups by going through all the internal nodes of T_0. Similarly, the jobs within each group $\mathcal{G}_i$ are also given as the leaf nodes of an arbitrary tree structure T_i, and the processors allocated to $\mathcal{G}_i$ need to be distributed to the jobs through the internal nodes of T_i. Figure 4(a) shows an example of such a multi-level hierarchy, where five job groups are organized as a tree with height 3 and the groups have their own tree hierarchies with possibly different heights. The objective is to minimize the total response time of all job groups, or the *group response time* $H(\mathcal{G}) = \sum_{i=1}^{m}(c_i - r_i)$, where r_i denotes the release time of group $\mathcal{G}_i$ and c_i denotes its completion time. Again, we assume that all jobs in a job group are released at the same time and a group is considered completed when all jobs in the group are completed.

Compared to the two-level scheduling model, this multi-level model places arbitrary tree structures between the source of the processors and the job groups as well as within each job group. Such an extension makes the hierarchical scheduling model applicable to an even wider range of *resource management problems*. Furthermore, the resources in these problems need not be limited to processor or computing resources, but can also refer to financial, inventory, or production resources, which need to be deployed through an organizational hierarchy to several groups or divisions, each one of which may in turn have its own internal hierarchy to distribute the resources. To minimize the makespan of all jobs in this hierarchical model, an $O(1)$-competitive algorithm was proposed in [5] based on a feedback-driven adaptive policy. In the next subsection, we will present a both fair and efficient solution for the more general objective of group response time by extending our previous results.

6.2 The FAIR∘EFFICIENT Solution

Our solution for the multi-level hierarchical scheduling model consists of two algorithms, namely, FAIR and EFFICIENT, which are responsible for guaranteeing the fairness among the job groups and the efficiency within each job group, respectively. We call the composite solution FAIR∘EFFICIENT.

FAIR Algorithm

The FAIR algorithm distributes the P processors to all active job groups at any time t, and as with EQUI, it gives each active job group an equal share, i.e., P/m_t processors, where m_t denotes the number of active job groups at time t. To achieve this in the presence of a tree hierarchy, FAIR first aggregates the number of active job groups through all the internal nodes of T_0 up the tree and then distributes the processors proportionally down the tree based on the active group count in each subtree.

Specifically, any active job group $\mathcal{G}_i$ at time t reports a desire of 1 to its parent node, i.e., $d(\mathcal{G}_i, t) = 1$, and

the desire $d(v,t)$ of an internal node $v \in T_0$ of the tree at time t is calculated by

$$d(v,t) = \sum_{u \in C_v} d(u,t), \tag{11}$$

where C_v denotes the set of immediate children of node v. Apparently, $d(v,t)$ represents the number of active job groups under node v at time t. After node v receives its processor allocation $a(v,t)$ from its parent, the share to be passed down to any of its child $w \in C_v$ is then based proportionally on the number of active groups in w, i.e.,

$$a(w,t) = \frac{d(w,t)}{\sum_{u \in C_v} d(u,t)} \cdot a(v,t). \tag{12}$$

According to the above proportional allocation policy, it is straightforward to see by induction that the number of processors allocated to each node $v \in T_0$ is given by $a(v,t) = P \cdot \frac{d(v,t)}{m_t}$. Hence, each active job group $\mathcal{G}_i$ (or leaf node of T_0) with $d(\mathcal{G}_i,t) = 1$ will get exactly $a(\mathcal{G}_i,t) = P/m_t$ processors, making FAIR equivalent to EQUI in terms of the processor allocation among the active job groups. Similarly to EQUI, FAIR only reallocates the processors whenever a new job group enters the system or an existing job group completes and leaves the system.

EFFICIENT Algorithm

The EFFICIENT algorithm is responsible for distributing the $a(\mathcal{G}_i,t)$ processors received by any active job group $\mathcal{G}_i$ among its active jobs. To interact directly with the jobs in $\mathcal{G}_i$, EFFICIENT can use any auxiliary scheduler X with provable efficiency to calculate the jobs' processor desires. As stated in Condition 1, the efficiency of X is then established by Inequalities (9) and (10) in terms of scheduling each individual job. To distribute the processors to the jobs, EFFICIENT first aggregates the jobs' processor desires computed by X up the tree T_i through its internal nodes according to Eq. (11). Then, any allocator Y that satisfies Condition 2, such as DEQ or CP, can be used to allocate the processors at each internal node of the tree according to the desires of its children. Under such a policy, any job (or node) will be guaranteed not to receive more processors than what it desires according to the conservative property of Condition 2. In addition, if any job (or node) is deprived, then the parent node must have distributed all of its processors according to the non-idle property of Condition 2. This implies that the parent and all the ancestors of the job must also be deprived, and hence all the $a(\mathcal{G}_i,t)$ processors received by the job group must be distributed to the active jobs, i.e., none of the processors is left idle.

Therefore, our solution FAIR∘EFFICIENT with a probably-efficient scheduler X and a conservative and non-idle allocator Y effectively reduces the multi-level hierarchical scheduling problem to the simple two-level scheduling problem. The solution for the two-level problem is an EQUI∘XY' algorithm, where the allocator Y' also satisfies both properties given in Condition 2 like Y does.[3] Figure 4(b) shows the reduced instance for the example shown in Figure 4(a), where the scheduling hierarchy is flattened by the desire aggregation and processor allocation schemes of the FAIR∘EFFICIENT solution.

7 Empirical Evaluations

In this section, we conduct simulations to empirically evaluate the performance of three non-clairvoyant algorithms we described in this paper for the two-level scheduling model, and they are EQUI∘AGDEQ, EQUI∘EQUI and EQUI∘ACDEQ. The objective is to verify our theoretical analysis, and to show that feedback-driven algorithms indeed achieve superior set response time and have better system efficiency.

7.1 Simulation Setup

We simulate a system with 64 processors, and the workload is generated based on a malleable job model [4], which creates synthetic parallel jobs with various internal parallelism structures. (See [4] for more details on the workload generation.) Taking a generated job sequence, we group consecutively released jobs together to form job sets. Both number of job sets and number of jobs in a job set are varied from 5 to 100 at an increment of 5 each time, so the total number of jobs in the system ranges from 25 to 10000. To make sure that all jobs in a job set are batch released, we adjust the release time of each job that belongs to a job set to when the first job of the set is released. Following the empirical advice in [1, 11], the utilization and responsiveness

[3] However, Y' may behave differently from Y in terms of the number of processors allocated to each active job for the two-level scheduling problem.

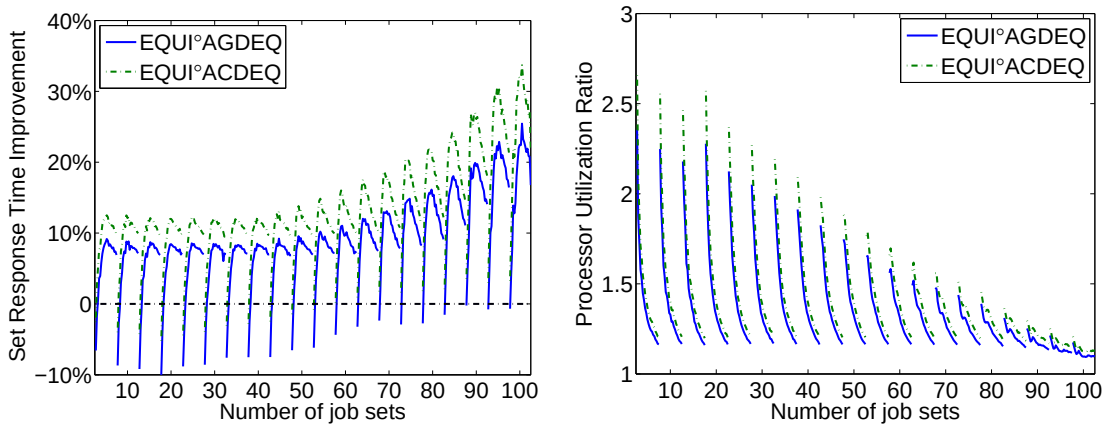

Figure 5: (a) Improvement of set response time by EQUI∘AGDEQ and EQUI∘ACDEQ over EQUI∘EQUI; (b) Normalized processor utilization of EQUI∘AGDEQ and EQUI∘ACDEQ relative to EQUI∘EQUI.

parameters of the A-GREEDY scheduler are set to be $\delta = 0.8$ and $\rho = 2$, respectively. The overhead incurred in the reallocation of the processors is ignored in the simulation.

7.2 Simulation Results

Figure 5(a) shows the improvements of the two feedback-driven algorithms over EQUI∘EQUI in terms of the set response time. First, we observe that when the number of job sets is fixed, the relative set response time is closely related to the number of jobs within each set. In particular, EQUI∘AGDEQ and EQUI∘ACDEQ outperform EQUI∘EQUI when there is a moderate to large number of jobs in each set, which is due to the better efficiency of the feedback-driven algorithms. Only when each job set contains a small number of jobs, the performance of EQUI becomes better, since the load in each job set is light in this case and nearly all jobs could be easily satisfied even by the oblivious strategy of EQUI at the job level. We can see that the advantage of the feedback-driven algorithms is more obvious when the overall load of the system (number of job sets) increases. The reason is because as each job set receives fewer processors, it is more important to be able to efficiently allocate the processors at the job level, especially when the load within each job set also increases. The results show that the peak performance improvements range from 10-30% compared to EQUI∘EQUI, depending on the load in our simulation. In addition, the performance of EQUI∘ACDEQ is slightly better than that of EQUI∘AGDEQ due to the more stable desire calculation strategy of the A-CONTROL scheduler [25].

Figure 5(b) shows the normalized processor utilizations of the two feedback-driven algorithms, which are always better than that of EQUI∘EQUI under all system loads. When there are very few jobs in each set, EQUI∘EQUI has particularly bad utilization, since it is blind to the jobs' resource requirements and thus wastes a lot of processor cycles. In fact, Figure 5(a) shows that EQUI∘EQUI achieves better set response time in this case at the cost of poor processor utilization. With increases in the system load, the advantage of the feedback-driven algorithms becomes smaller, as more processors are effectively allocated by EQUI∘EQUI when more jobs are present in the system. Even in this case, the utilization advantage is still around 10-20% in our simulation. The results confirm that the feedback-driven algorithms, which take advantage of the parallelism correlations of the jobs, indeed achieve better efficiency than EQUI∘EQUI in terms of processor utilizations.

8 Conclusion

In this paper, we have studied a two-level scheduling model to minimize the response time for multiple sets of parallel jobs on multiprocessor systems. We proposed online adaptive scheduling algorithms that achieve fairness and efficiency at the job-set level and the job level, respectively. Both theoretical analysis and simulation results demonstrate that our algorithms provide improved performance as compared to an existing scheduler that exhibits only fairness but not efficiency. Moreover, we provided a generalized analysis framework for a family of scheduling algorithms with provable efficiency and desirable allocation properties. Finally, we considered a multi-level hierarchical scheduling problem and proposed a both fair and efficient solution that effectively reduces it to the two-level scheduling model.

References

[1] K. Agrawal, Y. He, W.-J. Hsu, and C. E. Leiserson. Adaptive scheduling with parallelism feedback. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, New York, USA, 2006.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[3] A. Borodin, and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, USA, 1998.

[4] Y. Cao, H. Sun, W.-J. Hsu, and D. Qian. Malleable-Lab: a tool for evaluating adaptive online schedulers on malleable jobs. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, Pisa, Italy, 2010.

[5] Y. Cao, H. Sun, D. Qian, and W. Wu. Scalable hierarchical scheduling for multiprocessor systems using adaptive feedback-driven policies. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, Taipei, Taiwan, 2010.

[6] J. Corbalán, X. Martorell, and J. Labarta. Performance-driven processor allocation. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):599–611, 2005.

[7] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235(1):109–141, 2000.

[8] J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *Journal of Scheduling*, 6(3):231–250, 2003.

[9] J. Edmonds, and K. Pruhs. Scalably scheduling processes with arbitrary speedup curves. *ACM Transactions on Algorithms*, 8(3):28:1–28:10, 2012.

[10] D. G. Feitelson. Job scheduling in multiprogrammed parallel systems. *IBM Research Report RC19790(87657) 2nd Revision*, 1997.

[11] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1263–1279, 2008.

[12] Y. He, H. Sun, and W.-J. Hsu. Improved results for scheduling batched parallel jobs by using a generalized analysis framework. *Journal of Parallel and Distributed Computing*, 70(2):173–182, 2010.

[13] B. Kalyanasundaram, and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.

[14] Y.-K. Kwok, and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[15] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

[16] M. McNett, D. Gupta, A. Vahdat, and G.M. Voelker. Usher: an extensible framework for managing clusters of virtual machines. In *Proceedings of the Large Installation System Administration Conference*, Dallas, USA, 2007.

[17] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130(1):17-47, 1994.

[18] K. Pruhs. Competitive online scheduling for server systems. *Performance Evaluation Review*, 34(4):52–58, 2007.

[19] J. Robert, and N. Schabanel. Non-clairvoyant batch set scheduling: fairness is fair enough. In *Proceedings of the European Symposium on Algorithms (ESA)*, Eilat, Israel, 2007.

[20] J. Robert, and N. Schabanel. Pull-based data broadcast with dependencies: be fair to users, not to items. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, New Orleans, USA, 2007.

[21] S. Sen. Dynamic processor allocation for adaptively parallel jobs. Master's thesis, Massachusetts Institute of technology, 2004.

[22] M. Stillwell, F. Vivien, and H. Casanova. Dynamic fractional resource scheduling versus batch scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 23(3):521–529, 2012.

[23] R. Sudarsan, and C. J. Ribbens. Design and performance of a scheduling framework for resizable parallel applications. *Parallel Computing*, 36(1):48–64, 2010.

[24] H. Sun, Y. Cao, and W.-J. Hsu. Competitive two-level adaptive scheduling using resource augmentation. In *Proceedings of the Workshops on Job Scheduling Strategies for Parallel Processing (JSSPP)*, Rome, Italy, 2009.

[25] H. Sun, Y. Cao, and W.-J. Hsu. Efficient Adaptive scheduling of multiprocessors with stable parallelism feedback. In *IEEE Transactions on Parallel and Distributed Systems*, 22(4), 594–607, 2011.

[26] H. Sun, Y. Cao, and W.-J. Hsu. Fair and efficient online adaptive scheduling for multiple sets of parallel applications. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Tainan, Taiwan, 2011.

[27] H. Sun, Y. He, and W.-J. Hsu. Speed scaling for energy and performance with instantaneous parallelism. In *Proceedings of the International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS)*, Rome, Italy, 2011.

[28] H. Sun, and W.-J. Hsu. Adaptive B-Greedy (ABG): a simple yet efficient scheduling algorithm. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Miami, USA, 2008.

[29] A. Tucker, and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *ACM SIGOPS Operating Systems Review*, 23(5):159–166, 1989.

[30] J. B. Weissman, L. R. Abburi, and D. England. Integrated scheduling: the best of both worlds. *Journal of Parallel and Distributed Computing*, 63(6):649–668, 2003.