Non-migratory Multiprocessor Scheduling for Response Time and Energy

Tak-Wah Lam, Lap-Kei Lee, Isaac K. K. To, and Prudence W. H. Wong

Abstract—Energy usage has been an important concern in recent research on online job scheduling, where processors are allowed to vary the speed dynamically so as to save energy whenever possible. Providing good quality of service such as response time (flow time) and conserving energy are conflicting objectives. An interesting problem for scheduling is how to optimize an economic tradeoff of flow time and energy. To this end, the past two years have witnessed significant progress in the single-processor setting, and online algorithms with performance close to optimal have been obtained. In this paper we extend the study of optimizing the tradeoff between flow time and energy to the multi-processor setting. We devise and analyze a simple non-migratory online algorithm that makes use of the classified-round-robin (CRR) strategy to dispatch jobs. Even in the worst case, its performance is within $O(\log P)$ times of the optimal migratory offline algorithm, where P is the ratio of the maximum job size to the minimum job size. Technically speaking, this online result stems from a non-trivial solution to an offline problem of eliminating migration, which is also interesting by itself.

Index Terms—Analysis of Algorithms and Problem Complexity, Sequencing and scheduling, Online computation, Energy-aware systems.

1 Introduction

ENERGY consumption has become a key issue in the design of modern processors. A popular technology to reduce energy usage is *dynamic speed scaling* (see, e.g., [10], [21], [34]), where the processor can vary its speed dynamically. Running a job at a slower speed is more energy efficient, yet it takes longer time and may affect the performance. In the past few years, a lot of effort has been devoted to revisiting classical scheduling problems with dynamic speed scaling and energy usage taken into consideration (e.g., [1], [2], [7], [8], [11], [12], [23], [31], [36]; see [22] for a survey). The challenge basically arises from the conflicting objectives of providing good "quality of service" (QoS) and conserving energy.

One commonly used QoS measurement for scheduling jobs on a processor is the average response time, or equivalently, the total flow time (the latter is more popular in the literature of online scheduling). In this paper we investigate online scheduling algorithms for multi-processors, that take flow time and energy usage into consideration. Details are as follows.

1.1 Models and Previous Work

Jobs with arbitrary size are released at unpredictable times. A scheduling algorithm has to select, at any time, a job to execute. Preemption is allowed, and a preempted job can be resumed at the point of preemption. The flow time (or response time) of a job is the time elapsed since it arrives until it is completed.

Tak-Wah Lam and Lap-Kei Lee are with the Department of Computer Science, University of Hong Kong, Hong Kong. Email: {twlam,lklee}@cs.hku.hk; research is partially supported by a Hong Kong GRF Grant.
Isaac To and Prudence Wong are with the Department of Computer Science, University of Liverpool, UK. Email: {isaacto,pwong}@liverpool.ac.uk; research is partially supported by EPSRC Grant EP/E028276/1.

Online algorithms and competitive analysis. Both offline and online scheduling have been studied extensively. An *offline algorithm* knows the complete job sequence in advance, which is not possible in many practical problems. An *online algorithm*, in contrast, makes scheduling decisions based on jobs released so far, and thus is more applicable to reality. We analyze the performance of online algorithms using competitive analysis, i.e., a worst-case comparison between an online algorithm and the optimal offline algorithm. Formally speaking, given a cost function to minimize, such as total flow time, an online algorithm is said to be *c-competitive* if for any input, the cost incurred is never more than c times the cost required by an optimal offline algorithm. Note that the competitive ratio is a worst-case performance guarantee, and it does not depend on job distribution. For more details about competitive analysis, Borodin and El-Yaniv's book [9] is a good reference.

Speed models. The theoretical study of energy-efficient scheduling was initiated by Yao, Demers and Shenker [36]. They studied a model in which the processor incurs an energy of s^α per unit time when running at speed s, where α is typically 2 or 3 [10], [27]. In the *infinite speed model* [36], a processor can run at any speed between 0 and ∞. A more realistic model, known as the *bounded speed model* [12], imposes a bound T on the maximum allowable speed. In this paper, we focus on the bounded speed model.

Objective: Flow time and energy. The objectives flow time and energy are orthogonal. To better understand the tradeoff between them, Albers and Fujiwara [1] proposed combining the dual objectives into a single one of minimizing the sum of total flow time and energy.[1] The intuition is that, from an economic viewpoint, flow time and energy can each be measured in money terms; thus it can be assumed that users are willing to pay one unit of energy to reduce a certain units (say, ρ units) of flow time. A large value of ρ means that energy is more of a concern; on the other hand, if $\rho = 0$, the problem reduces to the traditional flow time scheduling. In general, the objective is to optimize the total flow time plus ρ times the energy used. $O(1)$-competitive algorithms have been proposed in both infinite speed [1], [8] and bounded speed model [6]. In the latter, a processor with maximum speed $(1+\epsilon)T$ is required (more details in Section 1.3).

Multi-processor scheduling for flow time and energy. When energy was not a concern, flow time scheduling on multi-processors running at fixed speed was an interesting problem by itself (e.g., [3], [4], [14], [15], [26], [29]). In this setting, jobs remain sequential in nature and cannot be executed by more than one processor in parallel. Different online algorithms have been proposed for the migratory, non-migratory, and immediate dispatching models, respectively. All of them are $\Theta(\log P)$-competitive, where P is the ratio of the maximum job size to the minimum job size.

In this paper, we extend the study of minimization of flow time plus energy to the setting with $m \geq 2$ processors. This extension is not only of theoretical interest, as modern processors adopt multi-core technology (dual-core and quad-core are getting common). A multi-core processor is essentially a pool of parallel processors. To make our work more meaningful, we aim at schedules that do not require job migration among processors. In practice, migrating jobs requires overheads and is avoided in many applications.

We defer the discussion of some practical considerations to Section 8, including some other energy models considered in the literature. We show that the model we adopted, though simpler, is general enough so that our results can be adapted to many other models easily.

1.2 New results and Ideas

The online algorithm CRR-SB and its performance. To balance the energy usage of multiple processors, it is natural to consider some kind of round-robin strategy to dispatch jobs. Typical examples include IMD[2] for flow time scheduling [3] and CRR for energy-efficient deadline scheduling[3] in the infinite speed model [2]. The main contribution of this paper is to apply CRR (classified round robin) for optimizing flow time plus energy and give a non-trivial analysis of its performance. Unlike [2], we apply CRR according to the job size rather than the job density. Specifically, we define the dispatching policy of CRR based on the following notion of *classes*. A job is said to be in class k if its size is in the range $(2^{k-1}, 2^k]$. Intuitively, CRR handles different classes independently; jobs of the same class are dispatched (upon their arrival) to the m processors using a round-robin strategy. Note that CRR is similar to IMD in the sense that both algorithms divide jobs into classes according to their size, but IMD is slightly more complicated in job dispatching. For minimizing flow time alone, IMD is known to be $O(\log P)$ competitive [3]; yet no similar result is known for CRR.

Jobs that are dispatched by CRR to the same processor can be scheduled by using an algorithm that minimizes flow time plus energy on a single processor. In particular, we schedule jobs using the online algorithm SRPT (Shortest Remaining Processing Time), and scale the speed based on the Bansal-Pruhs-Stein (BPS) algorithm [8]. Roughly speaking, the speed of each processor would be determined dynamically as the total fractional weight of jobs raised to the power of $1/\alpha$, where the fractional weight of a job is simply the unfinished fraction of the work of the job.

In analyzing the performance of the resulting algorithm, denoted as CRR-SRPT-BPS or CRR-SB in short, we focus on the bounded speed model and compare it against the optimal migratory offline algorithm using maximum speed T. We list below the competitiveness of CRR-SB for minimizing flow time plus energy.

- For jobs restricted to power-of-2 size, CRR-SB using processors with maximum speed T is $O(\alpha \log P)$-competitive.
- For jobs of arbitrary size, CRR-SB still has a similar performance when using processors with slightly higher maximum speed. Precisely, given any $\epsilon > 0$, CRR-SB using processors with maximum speed $(1+\epsilon)T$ is $(O(\alpha \log_{1+\epsilon} P) + 2\alpha(1+\epsilon)^{\alpha})$-competitive.

Our analysis can also be applied to the infinite speed model, though it is of less interest. In this case CRR-SB is $(O(\gamma \log P) + \gamma 2^{\alpha})$-competitive, where $\gamma = \max\{2, \frac{2(\alpha-1)}{\alpha-(\alpha-1)^{1-1/(\alpha-1)}}\} < 2\alpha$.

Lower bound. For minimizing flow time on multi-processors running at a fixed speed, Leonardi and Raz [26] showed that any online scheduling algorithm (without extra speed) is $\Omega(\log P)$-competitive. This lower bound can be easily adapted to the problem of minimizing flow time plus energy in the bounded speed model (see Section 7). That is, any online algorithm using speed scaling with maximum speed T is $\Omega(\log P)$-competitive for minimizing flow time plus energy. This lower bound remains valid even if jobs are all power-of-2 size (because the original proof in [26] uses only such jobs). In other words, CRR-SB when scheduling jobs of power-of-2 size

1. Sum of objectives are common in bi-objective optimizations, e.g., TCP acknowledgement problem [18], [25] with sum of acknowledgement cost and acknowledgement delays as objective, network design problem [20] with total hardware and QoS costs, and the facility location problem [17] with facility installation and client service costs.

2. IMD divide jobs into classes according to their sizes and dispatches a job to the processor with the smallest accumulated work of jobs of the corresponding class.

3. In [2], CRR divides jobs into classes according to their "density" (defined as the job size divided by the difference between deadline and release time), and dispatch jobs of the same class to the processors in a round-robin fashion.

achieves the best possible performance (up to a constant factor).

Eliminating migration offline. The analysis of CRR-SB relies on an offline result to eliminate migration, which is an interesting result on its own. The cost of eliminating migration has been investigated in the classical setting such as deadline scheduling [13], [24] and flow time scheduling [3], [4]. Our work extends along this line to take energy consumption into consideration. Precisely, given a migratory schedule $\mathcal{S}$ on m processors, we show how to construct a non-migratory schedule $\mathcal{N}$ such that the flow time plus energy incurred by $\mathcal{N}$ is at most $(O(\log P) + 2^\alpha)$ times of $\mathcal{S}$. More importantly, $\mathcal{N}$ dispatches jobs to processors in the same way as CRR. The last property is the key for deriving the online result and is difficult to attain. In fact, we can exploit existing results on flow time [3], [4] to eliminate migration with energy preserved; yet it is difficult to have the resulting schedule to always follow a certain online algorithm to dispatch jobs.

The above offline result stems from a transformation of an arbitrary job set to an "*m-parallel*" job set, in which jobs are partitioned into batches, each with m jobs of identical release time and size. We then observe that for an m-parallel job set, any migratory m-processor schedule can be transformed into a "*trivially non-migratory*" schedule, defined as any schedule in which all processors have identical schedules and at any time, execute respectively the m different jobs in a batch. The whole argument involves three transformations, exploiting tricks that allow us to modify the release times backward and forward, while constructing charging schemes to account for the shuffling of job schedules required.

Organization of paper. In the remainder of this section, we review some related work. In Section 2, we give some definitions necessary for discussion. In Section 3, we present the online algorithm CRR-SB. Section 4 gives an overview of the transformations involved in eliminating migration in offline scheduling. In Section 5, we investigate the cost of eliminating migration in a schedule of parallel jobs. In Section 6, we do the same for any job set, which also completes the analysis of CRR-SB. Then we give the $\Omega(\log P)$ lower bound in Section 7. Finally, in Section 8, we show that our results can be applied to other energy models.

1.3 Related work

In traditional flow time scheduling of single processor, the processor is running at a fixed speed (and thus energy is not a concern). The objective of a scheduler is simply to minimize the total flow time of all jobs, and the online algorithm SRPT (shortest remaining processing time) is known to produce an optimal (i.e., 1-competitive) schedule [5].

Single processor scheduling for flow time and energy. Pruhs et al. [30] were the first to consider flow time and energy together. They studied offline scheduling for minimizing the total flow time with a given amount of energy. They gave a polynomial time optimal algorithm for the special case when jobs are of unit size. However, in the online setting, this problem does not admit any constant competitive online algorithm even if jobs are of unit size [8].

For the objective total flow time plus ρ times energy, Albers and Fujiwara [1] considered the infinite speed model and presented the first online algorithm that is $O(1)$-competitive for jobs of unit size (precisely, the competitive ratio is $8.3e(\frac{3+\sqrt{5}}{2})^\alpha$, which is approximately 404 if $\alpha = 3$). This result was recently improved by Bansal et al. [8], who considered jobs with arbitrary size and presented an $O(1)$-competitive online algorithm. Precisely, the competitive ratio is $\mu_\epsilon \gamma$, where $\gamma = \max\{2, \frac{2(\alpha-1)}{\alpha-(\alpha-1)^{1-1/(\alpha-1)}}\}$ and $\mu_\epsilon = \max\{(1+1/\epsilon), (1+\epsilon)^\alpha\}$) for any $\epsilon > 0$. For example, suppose $\alpha = 3$, then choosing $\epsilon = 0.466$ would give a competitive ratio of 7.94.

Bansal et al. [6] adapted the previous results on minimizing flow time plus energy [8] to the bounded speed model. For jobs of arbitrary size, they gave a $(2\mu_\epsilon(\alpha + o(1))/\ln\alpha)$-competitive algorithm that uses a processor with maximum speed $(1+\epsilon)T$ for any $\epsilon > 0$.

Multi-processor scheduling for flow time and energy. The only previous work on multi-processors taking flow time and energy into consideration was by Bunde [11], which is about an offline approximation algorithm for jobs of unit size. As for online algorithms, no work has been known that takes flow time and energy into consideration.

Remarks. It is worth-mentioning that for other scheduling objectives (such as makespan and deadlines), the literature already contains several multi-processor results on dynamic speed scaling in the infinite speed model. In particular, Pruhs et al. [31] and Bunde [11] both studied offline algorithms for the makespan objective. Albers et al. [2] studied online algorithms for jobs with restricted deadlines. There are also some experimental work on different variants of energy-efficient deadline scheduling using dynamic speed scaling [16], [32], [33], [35], [37].

2 Preliminaries

Definitions and notations. Given a job set J, we want to schedule J on a pool of $m \geq 2$ processors. Jobs are sequential in nature and cannot be executed by more than one processor in parallel. All processors are identical and a job can be executed in any processor. Preemption is allowed and a preempted job can be resumed at the point of preemption. We differentiate two types of schedules: a migratory schedule can move partially-executed jobs from one processor to another processor without any penalty, and a non-migratory schedule dispatches each job to one of the m processors and runs the job entirely in that processor. In this paper the two types of schedules are usually represented by the symbols $\mathcal{S}$ and $\mathcal{N}$, respectively.

We use $r(j)$ and $p(j)$ to denote respectively the release time and work requirement (or size) of job j. The time required to complete a job j using a processor with fixed speed s is $p(j)/s$. For a set J of jobs, we let $p(J) = \sum_{j \in J} p(j)$ be the total size of J, and let $P(J)$ (or simply P) denote the ratio of the largest job size to the smallest job size. Following [8], we define the *fractional weight* of a job j at a particular time to be $q/p(j)$, where q is the remaining work of j at the time of concern. Note that the fractional weight decreases from 1 to 0 in the course of executing j.

With respect to a schedule $\mathcal{S}$ of a job set J, we use *max-speed*$(\mathcal{S})$, $E(\mathcal{S}), \bar{F}(\mathcal{S})$, and $F(\mathcal{S})$ to denote the maximum speed, energy usage, total flow time, and total fractional flow time of $\mathcal{S}$, respectively. Note that $\bar{F}(\mathcal{S})$ is the sum, over all jobs, of the time since a job is released until it is completed, or equivalently, the integration over time of the number of unfinished jobs. On the other hand, $F(\mathcal{S})$ is defined as the integration over time of the total fractional weight of unfinished jobs. Obviously, $F(\mathcal{S}) \leq \bar{F}(\mathcal{S})$. Note that processor speed can vary dynamically and the time to execute a job j is not necessarily equal to $p(j)$. We use $X(j)$ to denote the *execution time* of job j (i.e., flow time minus waiting time), and define $X(\mathcal{S}) = \sum_{j \in J} X(j)$ to be the total execution time of $\mathcal{S}$.

Our primary concern is to minimize the flow time plus ρ times the energy used. Technically, ρ can be assumed to be one.[4] Then the objective is simply to optimize total flow time plus energy. Yet it is helpful to first analyze the fractional flow time plus energy. It is convenient to define $\bar{G}(\mathcal{S}) = \bar{F}(\mathcal{S}) + E(\mathcal{S})$ and $G(\mathcal{S}) = F(\mathcal{S}) + E(\mathcal{S})$. The following lemma shows a lower bound that $\bar{G}(\mathcal{S}) \geq p(J)$, irrelevant of the number of processors. This is an extension of a similar bound for unit-size jobs given by Bansal et al. [8].

Lemma 1: For any m-processor schedule $\mathcal{S}$ for a job set J, $\bar{G}(\mathcal{S}) \geq p(J)$.

Proof: Denote F as the flow time of a job j. The energy usage for j is minimized if j is run at speed $p(j)/F$ throughout, and hence it is at least $F(p(j)/F)^\alpha = p(j)^\alpha(1/F)^{\alpha-1}$. It remains to show that $F + p(j)^\alpha / F^{\alpha-1} \geq p(j)$. *Case 1.* If $F \geq p(j)$, the statement is obviously true. *Case 2.* If $F < p(j)$, $(p(j)/F)^{\alpha-1} > 1$ and $p(j)^\alpha/F^{\alpha-1} \geq p(j)$. Summing over all jobs, we obtain the desired lower bound. □

4. Given a job set I to be scheduled on processors with maximum speed T for minimizing total flow time plus ρ times the energy, we define another job set I' by scaling the work of each job j in I to $p(j) \times \rho^{1/\alpha}$ for minimizing total flow time plus energy. Then any schedule S' for jobs I' with maximum speed $T' = \rho^{1/\alpha}T$ can be transformed to a schedule S for jobs I with maximum speed T (by decreasing the speed by a factor of $\rho^{1/\alpha}$), and vice versa. The total flow time remains the same while energy usage of S' is equal to $(\rho^{1/\alpha})^\alpha = \rho$ times that of S. Therefore, any algorithm minimizing total flow time plus energy can be extended to minimize total flow time plus ρ times the energy, and the competitive ratio preserves.

Global critical speed and flow time in multi-processor schedules. To optimize flow time plus energy, it is useful to define the *global critical speed* to be $1/(\alpha-1)^{1/\alpha}$, which is first observed by Albers and Fujiwara [1] in the context of single-processor scheduling. Throughout this paper, we assume that if necessary, a multi-processor schedule can be transformed without increasing the flow time plus energy so that it never runs a job j at speed less than the global critical speed (see Lemma 2 below and the proof in the Appendix) and hence, $X(j) \leq (\alpha-1)^{1/\alpha} p(j)$, which is at most $1.322p(j)$ for all possible $\alpha \geq 1$.

Lemma 2: Given any m-processor schedule $\mathcal{S}$ for a job set J, we can construct an m-processor schedule $\mathcal{S}'$ for J such that $\mathcal{S}'$ never runs a job at speed less than the global critical speed and $\bar{G}(\mathcal{S}') \leq \bar{G}(\mathcal{S})$. Moreover, $\mathcal{S}'$ needs migration if and only if $\mathcal{S}$ does; and *max-speed*$(\mathcal{S}') \leq \max\{$*max-speed*$(\mathcal{S}), 1/(\alpha-1)^{1/\alpha}\}$.

Furthermore, we assume that the maximum speed T is at least the global critical speed. Otherwise, any multi-processor schedule including the optimal one would always run a job at the maximum speed. It is because when running a job below the global critical speed, the slower the speed, the more total flow time plus energy is incurred. In other words, the problem is reduced to minimizing flow time alone.

Critical speed and fractional flow time in single-processor schedules. In our analysis of non-migratory schedules, we need to focus on individual processors and analyze the fractional flow time for each processor. In this case we need to consider fractional weight and make a different assumption of the minimum speed and transformation. At any time t, we define the *critical speed* of a job j to be $(q/(\alpha-1)p(j))^{1/\alpha}$, where $q \leq p(j)$ denotes the remaining work of job j at time t. Note that the critical speed of j changes over time. We can show that a single-processor schedule can be transformed without increasing the fractional flow time plus energy so that it never runs a job at speed less than its critical speed (see Lemma 3 below and the proof in the Appendix).

Lemma 3: Given any single-processor schedule $\mathcal{N}$ for a job set J, we can construct another schedule $\mathcal{N}'$ for J such that $\mathcal{N}'$ never runs a job at speed less than its critical speed and $G(\mathcal{N}') \leq G(\mathcal{N})$. Moreover, *max-speed*$(\mathcal{N}') \leq \max\{$*max-speed*$(\mathcal{N}), 1/(\alpha-1)^{1/\alpha}\}$.

3 THE ONLINE ALGORITHM CRR-SB

In this section we present an online algorithm, called CRR-SB (for CRR-SRPT-BPS), which produces a non-migratory CRR-dispatching schedule for $m \geq 2$ processors, and state some properties of CRR-SB. As mentioned earlier, the analysis of CRR-SB stems from an offline result to eliminate migration. We state a theorem about this offline result (to be proved in later sections) and analyze the performance of CRR-SB based on this offline result.

CRR dispatching. We now define CRR formally. Recall that a job is said to be in class k if its size is in the range

$(2^{k-1}, 2^k]$. Jobs of the same class are dispatched (upon their arrival) to the m processors using a round-robin strategy, i.e., the i-th job of a class is dispatched to processor $(i \bmod m)$, and different classes are handled independently. The resulting schedule is called a *CRR-dispatching* schedule. Once dispatched to a processor, a job will be processed there entirely; thus CRR-dispatching is non-migratory in nature.

The intuition of using a CRR-dispatching schedule comes from an offline result on eliminating migration, which states that for any migratory schedule, there is a CRR-dispatching schedule such that the total flow time plus energy is $O(\log P)$ times that of the migratory schedule (see Theorem 9 below). Then, to devise an online algorithm that is competitive against the optimal migratory offline algorithm, we can first dispatch jobs using a CRR-dispatching policy, and then schedule jobs in each processor independently, in a way that is competitive in the single-processor setting. This is the approach to be used by CRR-SB; more specifically, jobs dispatched to each processor is scheduled using SRPT and the speed is determined based on the single-processor algorithm BPS [6], [8].

Below we review the BPS algorithm and define CRR-SB. Unless otherwise stated, logarithms are in base 2.

> **Algorithm BPS.** At any time t, let $w_a(t)$ be the total fractional weight of jobs. Use the speed $w_a(t)^{1/\alpha}$ if allowed, or the maximum allowable speed otherwise.[5] Select the job with the smallest size to execute, ties are broken by selecting partially finished jobs.
>
> **Algorithm CRR-SB.** Jobs of the same class are dispatched to the m processors using the CRR-dispatching policy. Jobs in each processor are scheduled independently. At any time, the job with the shortest remaining work (processing time) is selected for execution, and the speed to be used is determined by the current speed of the following simulated schedule: The size of each job j dispatched to this processor is rounded up to $2^{\lceil \log p(j) \rceil}$. Simulate BPS on these enlarged jobs to obtain a simulated schedule.

We use J' to denote the set of enlarged jobs with size rounded up to the nearest power of 2. Note that we only make use of the speed of the simulated schedule, the exact way it schedules jobs is ignored. In fact, this schedule probably over-schedules each job j (for $2^{\lceil \log p(j) \rceil}$ instead of $p(j)$ units of work). On the other hand, after finishing $p(j)$ units of work for a job j, CRR-SB immediately moves on to the next job. At any time, CRR-SB and the simulated schedule probably schedule different jobs.

Adapting the analysis of BPS. Before we analyze the performance of CRR-SB, we need to adapt the previously known performance of BPS. It is known that BPS performs well in minimizing fractional flow time plus energy [6].

Fact 4: [6] For minimizing fractional flow time plus energy on a single processor, BPS is 2α-competitive.

However, the bound does not translate directly to flow time plus energy (i.e., $\bar{G}$). To bound the flow time plus energy of BPS, we first make the following observations about BPS.

Property 5: By definition, a BPS schedule satisfies the following properties.

i) At any time, the schedule has at most one partially processed job of each size.
ii) The schedule never idles when there are unfinished jobs.
iii) The schedule always runs a job at speed at least its critical speed.

For any non-migratory schedule satisfying the above properties, we can upper bound its flow time in terms of its fractional flow time and total execution time (Lemma 6), and the latter can be further shown to be at most twice of the total size of all jobs (Lemma 7). Then it is easy to analyze the flow time incurred by BPS (Corollary 8).

Lemma 6: Consider any non-migratory schedule $\mathcal{N}$ for a job set J in which Property 5 (i) and (ii) hold in every processor. Then $\bar{F}(\mathcal{N}) \leq F(\mathcal{N}) + K \cdot X(\mathcal{N})$, where K is the number of different job sizes in J. (Recall that $X(\mathcal{N})$ is the total execution time of $\mathcal{N}$.)

Proof: As $\mathcal{N}$ is non-migratory, we can analyze $\bar{F}(\mathcal{N})$ and $F(\mathcal{N})$ by summing up the total flow time and fractional flow time of individual processors. Consider any processor i, let $\bar{F}_i(\mathcal{N})$ ($F_i(\mathcal{N})$) be the corresponding total (fractional) flow time. At any time, the number of unfinished jobs in a processor can exceed the total fractional weight of unfinished jobs by at most the number of partially processed jobs, which is at most K. Furthermore, whenever a processor is idle, there is no unfinished jobs to charge to $\bar{F}_i(\mathcal{N})$ and $F_i(\mathcal{N})$. Thus, $\bar{F}_i(\mathcal{N}) - F_i(\mathcal{N}) \leq K \sum_{j \in J_i} X(j)$, where J_i is the subset of jobs executed in processor i. Summing over all processors, we have $\bar{F}(\mathcal{N}) - F(\mathcal{N}) \leq K \sum_{j \in J} X(j)$. □

Lemma 7: Consider any single-processor schedule $\mathcal{N}$ for a job set J satisfying Property 5 (iii). Then for any job j, $X(j) \leq (\alpha/(\alpha-1)^{1-1/\alpha})p(j) \leq 2p(j)$; and $X(\mathcal{N}) \leq 2p(J)$.

Proof: Consider a job j running at its critical speed starting at time 0 until it completes at some time t_c. At a time t, let q be the remaining work of j. Then the speed at t is $-\frac{dq}{dt} = (\frac{q}{(\alpha-1)p(j)})^{1/\alpha}$. This implies

$$\int_0^{t_c} dt = \int_{p(j)}^{0} -((\alpha-1)\,p(j))^{1/\alpha} q^{-1/\alpha}\, dq \ ,$$

or equivalently, $t_c = \frac{\alpha\, p(j)}{(\alpha-1)^{1-1/\alpha}}$. Consider any single-processor schedule $\mathcal{N}$ of J that never runs a job at speed less than its critical speed. Then for any job j,

$$X(j) \leq (\alpha/(\alpha-1)^{1-1/\alpha})p(j) \leq 2p(j),$$

5. When the objective function is total flow time plus ρ times energy, the speed is set as $w_a(t)^{1/\alpha}/\rho^{1/\alpha - 1/\alpha^2}$.

where the last inequality is due to the fact that $\alpha/(\alpha-1)^{1-1/\alpha}$ is maximized when $\alpha = 2$.

Therefore, summing over all jobs j,

$$X(\mathcal{N}) = \sum_{j \in J} X(j) \le \sum_{j \in J} 2p(j) = 2p(J). \qquad \square$$

We apply Lemmas 6 and 7 to upper bound the flow time incurred by BPS.

Corollary 8: Consider scheduling a job set J with K distinct sizes on a single processor. Let $\mathcal{N}$ and $\mathcal{N}^*$ be the schedule of BPS and the optimal offline schedule for J, respectively. Then $\bar{G}(\mathcal{N}) \le 2\alpha\bar{G}(\mathcal{N}^*) + 2Kp(J)$.

Proof: By Lemmas 6 and 7, $\bar{G}(\mathcal{N}) = \bar{F}(\mathcal{N}) + E(\mathcal{N}) \le F(\mathcal{N}) + 2Kp(J) + E(\mathcal{N})$. By Fact 4, the latter is at most $2\alpha(F(\mathcal{N}^*) + E(\mathcal{N}^*)) + 2Kp(J)$. Since $F(\mathcal{N}^*) \le \bar{F}(\mathcal{N}^*)$, $\bar{G}(\mathcal{N}) \le 2\alpha\bar{G}(\mathcal{N}^*) + 2Kp(J)$. $\square$

Analysis of CRR-SB. As we mentioned before, the analysis of CRR-SB is based on an offline result to eliminate migration. We state the theorem formally below (the next three sections are devoted to proving this theorem).

Theorem 9: Given a job set J, let J' be the same set of jobs but with size rounded up to the nearest power of 2. Let $\mathcal{S}$ be a migratory schedule of J. Then there is a CRR-dispatching schedule $\mathcal{N}$ for J', which also defines a schedule for J, such that

i) if jobs in J have arbitrary size, then $\bar{G}(\mathcal{N}) \le (2^\alpha + 11.932\lceil\log P\rceil + 14.576)\bar{G}(\mathcal{S})$, and *max-speed*$(\mathcal{N}) \le 2 \times$ *max-speed*$(\mathcal{S})$;
ii) if all jobs in J are of power-of-2 size, then $J' = J$, and $\bar{G}(\mathcal{N}) \le (5.966\lceil\log P\rceil + 7.288)\bar{G}(\mathcal{S})$ and the maximum speed remains the same.

Furthermore, if we change the definition of classes in CRR such that class k includes jobs of size in the range $((1+\lambda)^{k-1}, (1+\lambda)^k]$ for any $\lambda > 0$, then Theorem 9 (i) would give a CRR-dispatching schedule $\mathcal{N}$ such that $G(\mathcal{N}) \le ((1+\lambda)^\alpha + 5.966(1+\lambda)\lceil\log_{1+\lambda} P\rceil + 7.288(1+\lambda))\bar{G}(\mathcal{S}^*)$, and *max-speed*$(\mathcal{N}) \le (1+\lambda) \times$ *max-speed*$(\mathcal{S})$.

Recall that CRR-SB schedules jobs using SRPT and the speed function is based on the BPS algorithm. The following lemma shows that if we focus on a particular speed function, using SRPT is the best way to schedule a job set on a single processor. The proof adapts the result on fixed-speed processor scheduling in [5].

Lemma 10: Consider a job set J and a speed function f. Among all single-processor schedules of J using speed function f, the schedule that selects jobs using SRPT has the minimum total flow time.

Proof: We prove the lemma by contradiction. Among all single-processor schedules of J using speed function f, suppose that $\mathcal{N}^*$ is the one with the minimum total flow time and that there is a time instance $\mathcal{N}^*$ does not follow the SRPT policy. Let t be the first such time and j_ℓ be the job running at t in $\mathcal{N}^*$. Let j_s be a job with the shortest remaining processing time at t. Consider all the time intervals starting from t that $\mathcal{N}^*$ runs j_ℓ or j_s. We modify the schedule in these time intervals such that j_s is run to completion before j_ℓ starts, using the same speed function f. At time t, j_s has a smaller remaining processing time than j_ℓ; thus the sum of their completion time, as well as the sum of flow time in the new schedule are strictly less than before. The execution of other jobs remains unchanged. Therefore, the new schedule has a smaller total flow time than $\mathcal{N}^*$, which contradicts the optimality of $\mathcal{N}^*$. $\square$

We are now ready to prove the performance of CRR-SB using Theorem 9, Corollary 8 and Lemma 10.

Theorem 11: CRR-SB is $((23.864\alpha+4)\lceil\log P\rceil + (29.152 + 2^{\alpha+1})\alpha + 4)$-competitive for minimizing flow time plus energy when using processors with maximum speed $2T$ (the comparison is made against an optimal migratory schedule with maximum speed T).

Proof: Let $\mathcal{S}^*$ be the optimal migratory schedule of a job set J. Let J' be the set of jobs in J but with the size rounded up to the nearest power of 2. By Theorem 9 (i), we can obtain a CRR-dispatching schedule $\mathcal{N}_1$ for J' such that $\bar{G}(\mathcal{N}_1) \le (2^\alpha + 11.932\lceil\log P\rceil + 14.576)\bar{G}(\mathcal{S}^*)$. Let $\mathcal{N}_2$ be another CRR-dispatching schedule for J' such that jobs dispatched to each processor are scheduled by BPS. J' contains at most $\lceil\log P + 1\rceil$ different job sizes. Applying Corollary 8 to individual processors, we can conclude that $\bar{G}(\mathcal{N}_2) \le 2\alpha\bar{G}(\mathcal{N}_1) + 2\lceil\log P + 1\rceil p(J')$. On the other hand, $p(J') \le 2p(J)$ and, by Lemma 1, $p(J) \le \bar{G}(\mathcal{S}^*)$. Therefore,

$$\begin{aligned}\bar{G}(\mathcal{N}_2) &\le 2\alpha(2^\alpha + 11.932\lceil\log P\rceil \\ &\qquad + 14.576)\bar{G}(\mathcal{S}^*) + 4\lceil\log P + 1\rceil\bar{G}(\mathcal{S}^*) \\ &\le ((23.864\alpha + 4)\lceil\log P\rceil \\ &\qquad + (29.152 + 2^{\alpha+1})\alpha + 4)\bar{G}(\mathcal{S}^*).\end{aligned}$$

Let $\mathcal{N}$ be the schedule of CRR-SB for J. To prove Theorem 11, it remains to show that $\bar{G}(\mathcal{N}) \le \bar{G}(\mathcal{N}_2)$. First, we note that $\mathcal{N}_2$ also defines a CRR-dispatching schedule for J (just pad each job j in J with enough idle time to act as a job of size $2^{\lceil\log p(j)\rceil}$). Furthermore, $\mathcal{N}$, by definition, uses the same speed function as $\mathcal{N}_2$. Since $\mathcal{N}$ is using SRPT, we can apply Lemma 10 to individual processors to conclude that $\bar{G}(\mathcal{N}) \le \bar{G}(\mathcal{N}_2)$. Theorem 11 then follows. $\square$

Similarly, we have the following theorem for jobs of power-of-2 sizes.

Theorem 12: If jobs are of power-of-2 sizes, CRR-SB is $((11.932\alpha + 2)\lceil\log P\rceil + 16.576\alpha + 2)$ competitive in minimizing total flow time plus energy. The maximum speed used is the same as the optimal migratory offline algorithm.

Proof: Note that $J' = J$ in this case. We can then repeat the same argument in the proof of Theorem 11 by applying Theorem 9 (ii) instead of (i) to obtain the theorem. $\square$

Again, if we change the definition of classes in CRR-SB to use a base of $1+\lambda$ (instead of 2) for any $\lambda > 0$, then Theorem 11 would imply that CRR-SB, when using processors with maximum speed $(1+\lambda)T$, has a competitive ratio $(2\alpha(1+\lambda)^\alpha + (11.932\alpha(1+\lambda) + 2(1+\lambda))\lceil\log_{1+\lambda} P\rceil +$

$14.576\alpha(1+\lambda)+2(1+\lambda))$ for scheduling jobs of arbitrary size.

4 Overview of Transformations to Eliminate Migration

In this section we give an overview of an offline method for transforming a given migratory schedule to a non-migratory schedule. Roughly speaking, the method involves eliminating migration in two types of schedules: schedule for a set of special jobs called parallel jobs, and schedule for a set of arbitrary jobs. This section states the lemmas and theorems related to these two steps (to be proved in Sections 5 and 6). More importantly, we explain how to apply these results to obtain the main theorem of transforming a migratory schedule to a non-migratory schedule. To ease discussion, Figure 1 is given as an overview of how various lemmas and theorems are applied.

We first recall some definitions to be used in the following sections. We consider a job set J and denote the maximum-minimum ratio of job sizes by P. A job set is said to be *m-parallel* if the jobs can be partitioned into batches, each with m jobs of identical release time and size. For an m-parallel job set, a *trivially non-migratory* schedule is defined as any schedule in which all processors have identical schedules and at any time, execute respectively the m different jobs in a batch.

The key ideas and steps involved in the transformation are as follows.

1) It is easy to convert J into an m-parallel job set J^* (see the procedure **Make_Parallel** below). And a migratory schedule for J would naturally define a migratory schedule for J^*.
2) More interestingly, the migratory schedule for J^* can be transformed to a trivially non-migratory schedule for J^*.
3) Finally, any trivially non-migratory schedule for J^* can be transformed to a schedule for J which is CRR-dispatching.

Furthermore, all transformations incur only a moderate increase in the flow time plus energy.

The procedure **Make_Parallel** is defined as follows. It transforms J to three job sets, each has at most $\lceil \log P + 1 \rceil$ distinct job sizes.

- ***J'*** : Same as J except that the size of each job j is raised to $2^{\lceil \log p(j) \rceil}$.
- ***J^+***: Jobs of the same size in J' are grouped into batches of m jobs, in the order of release time. The last batch may not be full. J^+ is the set of all jobs that are in a "full" batch. Let $J^- = J' - J^+$.
- ***J^**** : For each batch in J^+, we pick a job with the earliest release time as the *leader*, and change the release time of every other job to that of the leader. We use $r(j)$ and $r^*(j)$ to denote the original and the new release time of a job j. The resulting job set is denoted by J^*, which is m-parallel.

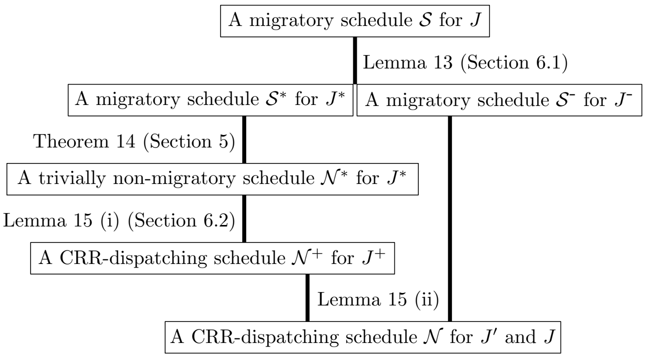

Fig. 1. Transforming a migratory schedule to a non-migratory schedule

The following lemmas and theorem define a sequence of transformations from a migratory schedule $\mathcal{S}$ of J to different intermediate schedules (for J^+, J^- and J^*) and eventually to a non-migratory CRR-dispatching schedule of J. See Figure 1 for a summary of these transformations. Each transformation consists of a few steps only; yet the analysis of the increase of flow time and energy is often quite involved. The details and proofs will be given in Sections 5 and 6. In the rest of this paper, we need to deal with different migratory and non-migratory schedules of the job sets J, J', J^+, J^- and J^*; it is noteworthy that their migratory schedules are always denoted by $\mathcal{S}, \mathcal{S}', \mathcal{S}^+, \mathcal{S}^-$, and $\mathcal{S}^*$, respectively, and their non-migratory schedules are denoted by $\mathcal{N}, \mathcal{N}', \mathcal{N}^+, \mathcal{N}^-$, and $\mathcal{N}^*$, respectively.

Lemma 13: Given a migratory schedule $\mathcal{S}$ for J, we can construct two migratory schedules $\mathcal{S}^*$ for J^* and $\mathcal{S}^-$ for J^- in such a way that $\bar{G}(\mathcal{S}^*) + \bar{G}(\mathcal{S}^-) \le 2^\alpha \bar{G}(\mathcal{S}) + 1.322 \lceil \log P + 2 \rceil p(J^+)$. Both $\mathcal{S}^*$ and $\mathcal{S}^-$ use no more than twice the maximum speed of $\mathcal{S}$.

The next transformation is the most non-trivial, it converts a migratory schedule for J^* to a trivially non-migratory schedule.

Theorem 14: Given a migratory m-processor schedule $\mathcal{S}^*$ for J^*, we can construct a trivially non-migratory schedule $\mathcal{N}^*$ for J^* such that $\bar{G}(\mathcal{N}^*) \le \bar{G}(\mathcal{S}^*) + 2 \lceil \log P + 1 \rceil p(J^*)$. Furthermore, *max-speed*$(\mathcal{N}^*) \le$ *max-speed*$(\mathcal{S}^*)$.

Recall that jobs in J^* may have their release time moved backward and thus we need another transformation of $\mathcal{N}^*$ to obtain a valid schedule for J^+ and then J.

Lemma 15: (i) Given a trivially non-migratory schedule $\mathcal{N}^*$ of J^*, we can construct a CRR-dispatching schedule $\mathcal{N}^+$ for J^+ such that $\bar{G}(\mathcal{N}^+) \le \bar{G}(\mathcal{N}^*) + 1.322 \lceil \log P + 1 \rceil p(J^+)$. Moreover, *max-speed*$(\mathcal{N}^+) \le$ *max-speed*$(\mathcal{N}^*)$. (ii) Together with a migratory schedule $\mathcal{S}^-$ for J^-, we can construct a CRR-dispatching schedule $\mathcal{N}$ for J' and J, such that $\bar{G}(\mathcal{N}) \le \bar{G}(\mathcal{N}^+) + \bar{G}(\mathcal{S}^-) + 1.322 \lceil \log P + 1 \rceil p(J')$. Moreover, *max-speed*$(\mathcal{N})$ is at most $\max\{$*max-speed*$(\mathcal{N}^+),$ *max-speed*$(\mathcal{S}^-)\}$.

With the above lemmas and theorem, we can prove the main result on eliminating migration, i.e., Theorem 9 (first stated in Section 3).

Proof of Theorem 9: Given a migratory schedule $\mathcal{S}$ of J, we can apply Lemma 13, Theorem 14, and Lemma 15 to obtain a CRR-dispatching schedule $\mathcal{N}$ for J' and J such that

$$\begin{aligned}\bar{G}(\mathcal{N}) &\le \bar{G}(\mathcal{N}^{+}) + \bar{G}(\mathcal{S}^{-}) + 1.322\lceil \log P + 1\rceil p(J') \\ &\le \bar{G}(\mathcal{N}^{*}) + 1.322\lceil \log P + 1\rceil p(J^{+}) + \bar{G}(\mathcal{S}^{-}) \\ &\quad + 1.322\lceil \log P + 1\rceil p(J') \\ &\le \bar{G}(\mathcal{S}^{*}) + 2\lceil \log P + 1\rceil p(J^{*}) \\ &\quad + 1.322\lceil \log P + 1\rceil p(J^{+}) + \bar{G}(\mathcal{S}^{-}) \\ &\quad + 1.322\lceil \log P + 1\rceil p(J') \\ &\le 2^{\alpha}\bar{G}(\mathcal{S}) + 2\lceil \log P + 1\rceil p(J^{*}) \\ &\quad + (2.644\lceil \log P\rceil + 3.966)p(J^{+}) \\ &\quad + 1.322\lceil \log P + 1\rceil p(J').\end{aligned}$$

Note that $p(J^{+})$, $p(J^{*})$ and $p(J')$ are at most $2p(J)$. By Lemma 1, $p(J) \le \bar{G}(\mathcal{S})$. Thus, $\bar{G}(\mathcal{N}) \le (2^{\alpha} + 11.932\lceil \log P\rceil + 14.576)\bar{G}(\mathcal{S})$. Furthermore, the maximum speed is at most double of the maximum speed of the optimal migratory schedule. Thus, Theorem 9 (i) holds.

Suppose that all jobs in J are of power-of-2 size. Then we can prove Lemma 13 without doubling the maximum speed and increasing the energy usage by a factor of 2^{α}; that is, $\bar{G}(\mathcal{S}^{*}) + \bar{G}(\mathcal{S}^{-}) \le \bar{G}(\mathcal{S}) + 1.322\lceil \log P + 2\rceil p(J^{+})$. Furthermore, $p(J^{+}) = p(J^{*}) \le p(J') = p(J)$. Thus we could improve the upper bound of $\bar{G}(\mathcal{N})$ to $(5.966\lceil \log P\rceil + 7.288)\bar{G}(\mathcal{S})$, and the maximum speed remains the same. Thus, Theorem 9 (ii) holds. □

5 Eliminating Migration in a Multi-Processor Schedule of Parallel Jobs

In this section we prove Theorem 14 (of Section 4) that transforms a migratory schedule $\mathcal{S}^{*}$ of an m-parallel job set J^{*} to a trivially non-migratory schedule $\mathcal{N}^{*}$ for J^{*} with a moderate increase in flow time plus energy. Let K denote the number of distinct job sizes in J^{*}, i.e., $K = \lceil \log P + 1\rceil$.

The transformation consists of two steps. Each step preserves the total fractional flow time plus energy. The first step makes use of an "averaging" technique to determine the speed and to distribute the workload among the processors, this results in a non-migratory and indeed trivially non-migratory schedule $\mathcal{N}_1^{*}$. The second step attempts to locally "tidy up" the schedule of each individual processor in $\mathcal{N}_1^{*}$ so that the total flow time of the resulting schedule $\mathcal{N}^{*}$ does not exceed its total fractional flow time too much (precisely, by at most $2Kp(J^{*})$). Then Theorem 14 follows. Details are as follows.

Step 1: Speed Averaging. The speed function of the new schedule $\mathcal{N}_1^{*}$ is determined as follows. At any time, every processor in $\mathcal{N}_1^{*}$ runs at the average speed of all processors of $\mathcal{S}^{*}$. That is, if the processors in $\mathcal{S}^{*}$ are at speed $s_1, s_2, \ldots, s_m$, respectively, then every processor in $\mathcal{N}_1^{*}$ runs at speed $\sum_{i=1}^{m} s_i/m$. Note that the "total" speed of $\mathcal{S}^{*}$ and $\mathcal{N}_1^{*}$ are the same. Since the energy function s^{α} is convex, the rate of energy consumption of $\mathcal{N}_1^{*}$ (i.e., $m\left(\sum_{i=1}^{m} s_i/m\right)^{\alpha}$) is at most that of $\mathcal{S}^{*}$ (i.e., $\sum_{i=1}^{m} s_i^{\alpha}$).

Work Averaging. Next we describe how $\mathcal{N}_1^{*}$ selects jobs for execution. In $\mathcal{S}^{*}$, jobs in a batch may have different progress. To ease our discussion, we divide $\mathcal{S}^{*}$ into consecutive time intervals at the moment when a batch of jobs is released or has just been completed. Let $\mathcal{I}$ be such a time interval. Within $\mathcal{I}$, suppose $\mathcal{S}^{*}$ has worked on some jobs of a batch B for a total of u units (note that the work done on each job of B may vary), then $\mathcal{N}_1^{*}$ would schedule each processor to work on u/m units of a different job of B in parallel. Note that the total work done on B is still u (though the progress of individual jobs might differ from $\mathcal{S}^{*}$). $\mathcal{S}^{*}$ might have worked on several batches within $\mathcal{I}$; at any time within $\mathcal{I}$, $\mathcal{N}_1^{*}$ uses the smallest-job-size-first (SJF) strategy to select the next batch for execution.

Analysis. At any time in an interval $\mathcal{I}$, the SJF strategy ensures that $\mathcal{N}_1^{*}$ gives priority to jobs that would give the biggest decrease of fractional weight; thus, $\mathcal{N}_1^{*}$ has a total fractional weight no more than that of $\mathcal{S}^{*}$. At the end of $\mathcal{I}$, $\mathcal{N}_1^{*}$ and $\mathcal{S}^{*}$ have performed the same amount of work for each batch, and they have the same total fractional weight. Applying the same argument to every time interval of $\mathcal{S}^{*}$, we conclude that at any time, the total fractional weight in $\mathcal{N}_1^{*}$ is no more than that of $\mathcal{S}^{*}$. Thus, the total fractional flow time of $\mathcal{N}_1^{*}$ is also no more than that of $\mathcal{S}^{*}$.

Step 2 (Tidying). We further transform $\mathcal{N}_1^{*}$ to $\mathcal{N}^{*}$ to reduce the total flow time. Recall that $\mathcal{N}_1^{*}$ has an identical schedule for all processors. The changes made in Step 2 are local to each processor, and all processors undergo the same changes.

(a) **Critical speed.** We ensure that at any time, $\mathcal{N}_1^{*}$ executes a job j at speed at least its critical speed (recall that this critical speed property can be enforced by invoking the transformation stated in Lemma 3 to each processor).

(b) **Minimizing partially processed jobs.** Next we want to ensure that in each processor, there is at most one partially processed job of each size. To obtain such a schedule, we consider all jobs of a particular size each time, and shuffle these jobs using "earliest release time first" strategy. The speed used at any time is not changed, so after shuffling, a job may be executed faster or slower.

(c) **Eliminating unnecessary idle time.** Finally, we want to "compact" the schedule of each processor so that it is never idle when there are unfinished jobs. To do so, we consider all the jobs executed in a processor in the order of release time. For each job j, we move its schedule as close as possible to its release time, filling out all the idle time. Note that we do not change the speed and the total time to execute j.

Analysis. Denote the schedule produced by Step 2 as

$\mathcal{N}^*$. By Lemma 3, Step 2(a) does not increase the total fractional flow time plus energy. Steps 2(b) and (c) do not change the energy usage. Furthermore, in Step 2(b), shuffling jobs of the same size among themselves does not alter the total fractional weight of these jobs. Thus the total fractional flow time is preserved. Step 2(c) could only decrease the total fractional flow time. Thus $\mathcal{N}^*$ does not increase the total fractional flow time plus energy.

Next, we consider the total flow time. Note that Step 2(a) produces a schedule that runs a job at speed no less than its critical speed. So by Lemma 7, its total execution time is at most $2p(J^*)$. Steps 2(b) and 2(c) do not change the total execution time; we conclude that $X(\mathcal{N}^*) \leq 2p(J^*)$. Applying Lemma 6 to $\mathcal{N}^*$, $\bar{F}(\mathcal{N}^*) \leq F(\mathcal{N}^*) + 2Kp(J^*)$. Hence $\bar{G}(\mathcal{N}^*) \leq G(\mathcal{N}^*) + 2Kp(J^*)$. Since $\mathcal{N}^*$ preserves the total fractional flow time plus energy of $\mathcal{S}^*$ and the fractional flow time is always upper bounded by flow time, we have $G(\mathcal{N}^*) \leq G(\mathcal{S}^*) \leq \bar{G}(\mathcal{S}^*)$, and $\bar{G}(\mathcal{N}^*) \leq \bar{G}(\mathcal{S}^*) + 2Kp(J^*)$. Theorem 14 is proved.

6 ELIMINATING MIGRATION IN A MULTI-PROCESSOR SCHEDULE OF ARBITRARY JOBS

In this section we show how to make use of the previous result on parallel jobs to derive a way to eliminate migration in a schedule for an arbitrary job set J. Recall that in Section 4, we have defined the job sets J', J^+, J^- and J^* from a given job set J. Two transformations will be presented in this section: a forward transformation of a migratory schedule for J to a migratory schedule for J^* (Section 6.1) and a backward transformation of a trivially non-migratory schedule for J^* to a CRR-dispatching schedule for J (Section 6.2).

6.1 Forward transformation of schedules: from J to J^*

In this section we show how to transform a migratory schedule $\mathcal{S}$ for J to a migratory schedule $\mathcal{S}^*$ for J^*. Recall that a job j in J^* has a bigger size ($2^{\lceil \log p(j) \rceil}$) and an earlier deadline ($r^*(j)$), and the desired lemma is as follows.

Lemma 13 *Given a migratory schedule $\mathcal{S}$ for J, we can construct two migratory schedules $\mathcal{S}^*$ for J^* and $\mathcal{S}^-$ for J^- in such a way that $\bar{G}(\mathcal{S}^*) + \bar{G}(\mathcal{S}^-) \leq 2^\alpha \bar{G}(\mathcal{S}) + 1.322 \lceil \log P + 2 \rceil p(J^+)$. Both $\mathcal{S}^*$ and $\mathcal{S}^-$ use no more than twice the maximum speed of $\mathcal{S}$.*

To construct $\mathcal{S}^*$ from $\mathcal{S}$, we makes use of two ideas:

- Speed up the schedule of each job j in $\mathcal{S}$ to counter the increase of size.
- Advance the schedule of each job j in view of its new release time $r^*(j)$.

Details are as follows. Below we assume that $\mathcal{S}$ always schedules every processor to run at speed at least the global critical speed (see Lemma 2).

Speed up the schedule. $\mathcal{S}$ naturally defines a schedule $\mathcal{S}'$ for J' as follows: Whenever $\mathcal{S}$ runs a job j at speed s, $\mathcal{S}'$ runs j at speed $\frac{2^{\lceil \log p(j) \rceil}}{p(j)} s$. Note that the maximum speed is at most doubled. Recall that $J' = J^+ \cup J^-$. Restricting $\mathcal{S}'$ with respect to J^+ and J^- gives two schedules $\mathcal{S}^+$ and $\mathcal{S}^-$, respectively. Note that $E(\mathcal{S}^+) + E(\mathcal{S}^-) \leq 2^\alpha E(\mathcal{S})$, and $\bar{F}(\mathcal{S}^+) + \bar{F}(\mathcal{S}^-) = \bar{F}(\mathcal{S})$.

$\mathcal{S}^+$ is also a valid schedule for J^*, though it might induce a larger total flow time (because jobs in J^* have earlier release time). To limit the increase in flow time, we modify $\mathcal{S}^+$ into a better schedule $\mathcal{S}^*$ for J^* as follows. Note that we will not change the energy used, i.e., $E(\mathcal{S}^*) = E(\mathcal{S}^+)$.

Advance the schedule. Tag the jobs in each group of J^+ with unique integers from 0 to $m-1$, 0 being the leader. The schedule of all leaders are not modified. We modify $\mathcal{S}^+$ in rounds, one for a particular job size (say, from the smallest to the largest). In each round, we consider groups of jobs in the order of the release time. Consider a job j of a particular group, which is tagged with $i > 0$. Let j' be its leader. To reduce the flow time of j, we use the following trick to advance the schedule of j using a single processor, in particular, processor labeled i. Let t_1 be the total amount of time over all processors that $\mathcal{S}^+$ executes j, and suppose that between the (original) release times of j' and j, i.e., $[r^*(j), r(j)]$ (recall that $r^*(j) = r(j')$), the current schedule of processor i has a total of $t_2 \geq 0$ units of idling time. If $t_1 < t_2$, we move the schedule of j to processor i only, occupying the earliest idle time starting from $r^*(j)$. The speed function for processing j remains unchanged. If $t_1 \geq t_2$, the schedule of j is left unmodified.

Analysis. The above modification guarantees that in the new schedule, the job j, whose release time has been set to $r^*(j)$, has only a moderate increase in waiting time, precisely, at most the sum of execution time of processor i during $[r^*(j), r(j)]$ and $X(j)$. This is proved in Lemma 16, which also shows that from J^+ to J^*, the total increase of waiting time (as well as flow time) over all jobs is at most $\lceil \log P + 2 \rceil X(\mathcal{S}^+)$ Since every processor run at speed at least the global critical speed (Lemma 2), $X(\mathcal{S}^+) \leq (\alpha - 1)^{1/\alpha} p(J^+) \leq 1.322 \lceil \log P + 2 \rceil p(J^+)$, and Lemma 13 follows.

Lemma 16: The increase in flow time caused by the transformation in Lemma 13, when transforming a migratory schedule $\mathcal{S}^+$ for J^+ to a schedule $\mathcal{S}^*$ for J^*, is at most $\lceil \log P + 2 \rceil X(\mathcal{S}^+)$.

Proof: Since the same speed is being used for all jobs, $X(\mathcal{S}^*) = X(\mathcal{S}^+)$. It remains to analyze the waiting time. This is done by bounding the change in waiting time of each modified job during each round. Let $\mathcal{S}_a$ and $\mathcal{S}_b$ be the schedules before and after a round in the transformation. Obviously there is no change in the waiting time for leaders. Consider a job j tagged with $i > 0$, with leader j'. We claim that the increase in waiting time is at most the sum of the execution time of j in $\mathcal{S}_a$ and the busy time of processor i in schedule $\mathcal{S}_a$ during $[r^*(j), r(j)]$. This is clear in the case where the scheduling of j is modified: indeed the waiting time of j in $\mathcal{S}_b$ is no more than latter term. In case where the scheduling of j is unmodified, the increase in waiting time is $r(j) - r^*(j)$, which equals to the amount of idle time plus busy time of processor i in schedule $\mathcal{S}_a$ during $[r^*(j), r(j)]$. Since the scheduling of j is not modified, the amount of idle time is at most the execution time of j in $\mathcal{S}_a$. So, the claim holds.

Summing over all jobs concerned for the modification from $\mathcal{S}_a$ to $\mathcal{S}_b$ for size 2^k jobs, the increase in waiting time is no more than the sum of $X(\mathcal{S}_a) = X(\mathcal{S}^+)$ and $\sum_{j \text{ is of size } 2^k} X(j)$. Summing over all the $\lceil \log P + 1 \rceil$ rounds, the waiting time is increased by no more than $\lceil \log P + 2 \rceil X(\mathcal{S}^+)$. □

6.2 Backward transformation of schedules: from J^* to J^+ and then to J

Recall that J^* is m-parallel. Suppose that we have used Theorem 14 to obtain a trivially non-migratory schedule $\mathcal{N}^*$ for J^*. Below we show how to transform $\mathcal{N}^*$ to a CRR-dispatching schedule $\mathcal{N}^+$ for J^+ (Lemma15 (i)). Then it is relatively easy to obtain a CRR-dispatching schedule for J', as well as for J (Lemma 15 (ii)). We first reiterate Lemma 15 of Section 4 before proving it.

Lemma 15 (i) *Given a trivially non-migratory schedule $\mathcal{N}^*$ of J^*, we can construct a CRR-dispatching schedule $\mathcal{N}^+$ for J^+ such that $\bar{G}(\mathcal{N}^+) \le \bar{G}(\mathcal{N}^*) + 1.322 \lceil \log P + 1 \rceil p(J^+)$. Moreover, max-speed($\mathcal{N}^+$) $\le$ max-speed($\mathcal{N}^*$).* (ii) *Together with a migratory schedule $\mathcal{S}^-$ for J^-, we can construct a CRR-dispatching schedule $\mathcal{N}$ for J' and J, such that $\bar{G}(\mathcal{N}) \le \bar{G}(\mathcal{N}^+) + \bar{G}(\mathcal{S}^-) + 1.322 \lceil \log P + 1 \rceil p(J')$. Moreover, max-speed($\mathcal{N}$) is at most $\max\{$max-speed($\mathcal{N}^+$), max-speed($\mathcal{S}^-$)$\}$.*

From J^* to J^+. Note that $\mathcal{N}^*$ may not be a valid schedule for J^+, since a job j in J^* has a release time $r^*(j)$ earlier than the release time $r(j)$ in J^+. We transform $\mathcal{N}^*$ to move the execution period of jobs so that the schedule becomes valid. The changes made are local to each processor, and all processors undergo the same changes. The following discussion focuses on the schedule of a processor x in $\mathcal{N}^*$. Without loss of generality, we assume that the jobs of the same size are processed in the order of release time in J^+. Furthermore, by Lemma 2, the processor runs at a speed at least the global critical speed.

Transformation. We focus on the schedule of a particular processor x in $\mathcal{N}^*$. The transformation runs in multiple rounds. Initially, R is the schedule of the processor x in $\mathcal{N}^*$. In each round, R is modified with respect to the jobs of a particular size. Recall that $r(j)$ and $r^*(j)$ denote the release time of a job j in J^+ and J^*, respectively. Let $j_1, j_2, \ldots, j_n$ be the jobs of size p running in processor x, where $r^*(j_1) \le r^*(j_2) \le \cdots \le r^*(j_n)$. We observe that $r^*(j_i) \le r(j_i) \le r^*(j_{i+1})$; the latter inequality is due to the fact that $r^*(j_{i+1})$ is actually the release time of the leader of j_{i+1} in J^+, which is at least $r(j_i)$. In other words, the period where R schedules j_{i+1} is a feasible period to schedule j_i in J^+. The transformation first removes the scheduling of $j_1, \ldots, j_n$ from R. Then, for each j_i from $i = 1$ to $n-1$, we schedule j_i to the first idle intervals of the schedule after $r(j_i)$, using the same speeds of j_{i+1} in R. Finally, we schedule j_n to the first idle intervals of the schedule after $r(j_n)$ using the same speeds of j_1 in R. The transformation is repeated for each job size. The final schedule obtained for all processors is $\mathcal{N}^+$.

As the transformation is local to each processor, $\mathcal{N}^+$ follows the way $\mathcal{N}^*$ dispatches jobs. Consider jobs in J^+ in the order of release time. Because $\mathcal{N}^*$ is trivially non-migratory, $\mathcal{N}^+$ dispatches every m jobs of the same size to different processors. Thus $\mathcal{N}^+$ is a CRR-dispatching schedule.

Analysis. In the transformation above, the maximum speed and energy of $\mathcal{N}^+$ remains the same as that of $\mathcal{N}^*$. Yet the analysis of the flow time is quite tricky. The increase in flow time is the increase in execution time plus waiting time of all jobs. Consider a particular processor and a particular job size p for which a schedule R is transformed to R_1. *Execution time:* The speed used by R_1 for the jobs is the same as that used by R, for a permutation of the jobs. Thus there is no change in total execution time for all jobs involved. *Waiting time:* According to the way we transform the execution period and speed of j_i, one can argue (by induction) that the start time (resp. completion time) of j_i in R_1 is no later than the start time (resp. completion time) of j_{i+1} in R (see Lemma 17). This property enables us to show that from $\mathcal{N}^*$ to $\mathcal{N}^+$, the increase of total flow time is at most $\lceil \log P + 1 \rceil X(\mathcal{N}^+)$ (see Lemma 17), which is at most $1.322 \lceil \log P + 1 \rceil p(J^+)$ (by Lemma 2). Then Lemma 15 (i) follows.

Lemma 17: The transformation in Lemma 15 transforms a trivially non-migratory schedule $\mathcal{N}^*$ for J^* to a non-migratory CRR-dispatching schedule $\mathcal{N}^+$ for J^+ such that $F(\mathcal{N}^+) \le F(\mathcal{N}^*) + \lceil \log P + 1 \rceil X(\mathcal{N}^+)$.

Proof: As mentioned before, it suffices to bound the waiting time among all jobs. Due to the tidying step in the course of constructing the trivially non-migratory schedule $\mathcal{N}^*$ of J^*, we can assume that the jobs of the same size are processed in the order of release time in J^+. Consider a round of the transformation that converts from schedule R to R_1 for a particular processor and a particular job size p. Note that the waiting time of a job not of size p does not change, so we focus on jobs of size p. Recall that the transformation is done by first removing all jobs $j_1, j_2, \ldots, j_n$ of size p from R. We use R_0 to denote this partial schedule. Let $c^*(j)$ and $c(j)$ denote the completion time of job j in R and R_1, respectively. We first observe the following property about the start time and completion time of jobs j_i in R and R_1.

Proposition: For $i = 1, \ldots, n-1$, j_i starts running (respectively, is completed) in R_1 at or before j_{i+1} starts running (respectively, is completed) in R.

The proposition can be proved by induction on i. Job j_i starts running in R_1 at either (1) the first idle time in R_0 at or after $c(j_{i-1})$, or (2) the first idle time in R_0 at or after $r(j_i)$, whichever larger. By induction, the time (1) is no later than the first idle time in R_0 at or after $c^*(j_i)$. By our assumption that jobs of the same size are scheduled in the order of release time, this is the first time that R can process job j_{i+1}. The time (2) is no later than the first idle time in R_0 at or after $r^*(j_{i+1})$ (since $r(j_i) \leq r^*(j_{i+1})$), which is obviously no later than the start time of j_{i+1}. So the start time of j_i in R_1 is no later than the start time of j_{i+1} in R. Now $c(j_i) \leq c^*(j_{i+1})$ follows immediately, since the speed function that R_1 uses for j_i is copied from the speed function that R uses for j_{i+1}.

To bound the increase in waiting time among $j_1, j_2, \ldots, j_n$, we characterize such increase, i.e., times which contribute to the waiting time of j_i in R_1 but not in R. They can be times when one of the followings happen.

- R_0 works on a job of size other than p when j_i is already completed in R but waiting in R_1. In this case the time must be during $[c^*(j_i), c(j_i)]$.
- R_0 is idle when j_{i-1} has already completed in R but is running in R_1. In this case the time must be during $[c^*(j_{i-1}), c(j_{i-1})]$.

In both cases, these times contributed to the waiting time are times when R_1 is running; furthermore, by the Proposition, such times do not overlap for different j_i. The increase in waiting time over all jobs is thus at most $X(R_1)$. Summing over all processors and all $\lceil \log P + 1 \rceil$ rounds, the lemma follows. $\square$

From J^+ and J^- to J' and J. Let us turn to the modification of the schedule to accommodate the jobs in J^- and to handle the reduction in job size when scheduling J.

To construct $\mathcal{N}$ in Lemma 15 (ii), we simply insert each job $j \in J^-$ into the given schedule $\mathcal{N}^+$. By definition, j has a release time later than jobs in J^+ of the same size, and would be dispatched by CRR after all these jobs. Suppose CRR dispatches j to processor i. Then we schedule j to processor i during the first idle periods after j's release time, using the speed function of j in $\mathcal{S}^-$. The total waiting time of j is at most the total execution time of processor i. After inserting all jobs in J^-, we obtain a CRR-dispatching schedule $\mathcal{N}'$ for J', with energy usage and maximum speed preserved. Since at most $\lceil \log P + 1 \rceil$ jobs in J^- are dispatched to each processor, the total increase in flow time is at most $\lceil \log P + 1 \rceil$ times the total execution time of all processors in $\mathcal{N}'$, which is at most $1.322 \lceil \log P + 1 \rceil p(J')$, by Lemma 2.

$\mathcal{N}'$ immediately implies a non-migratory schedule $\mathcal{N}$ for J with the same properties. $\mathcal{N}$ simply follows $\mathcal{N}'$ and is idle whenever the job to schedule is already completed. Thus, $\bar{G}(\mathcal{N}) \leq \bar{G}(\mathcal{N}')$.

7 Lower bound

We show a lower bound for minimizing total flow time plus energy when there is a maximum speed. The bound holds even if jobs are restricted to power-of-2 sizes, implying that CRR-SB is optimal in this setting. We start with a lower bound of $\Omega(\log P)$ given by Leonardi and Raz [26] on the competitive ratio of any migratory online algorithm for flow time scheduling on fixed-speed processors. Their proof is based on power-of-2 sized jobs. This proof can be adapted easily to show the following theorem.

Theorem 18: Any online (migratory) algorithm for minimizing flow time plus energy on multiple processors with a maximum speed has a competitive ratio of $\Omega(\log P)$. This holds even if jobs are restricted to have power-of-2 sizes.

Proof: Consider the case that $T = 1$, which is greater than the global critical speed when $\alpha > 2$. Let's consider a particular online algorithm. Let $\bar{F}_a$, F_a, and $\bar{G}_a$ be the flow time, fractional flow time, and flow time plus energy of the online algorithm, respectively. By the lower bound result in [26], there is some constant c that for any online algorithm (including the one which we are considering), we can find some input J consisting of jobs of power-of-2 sizes, such that $\bar{F}_a \geq (c \log P)\bar{F}_s$, where $\bar{F}_a$ is the flow time required by the online algorithm, and $\bar{F}_s$ is the minimum total flow time. Obviously, the minimum total flow time is achieved by running all processors at the maximum speed 1. The energy consumption of such a schedule is thus the time the processor is running, which is at most $\bar{F}_s$. We thus have a schedule of J with $\bar{G} \leq 2\bar{F}_s$, leading to that the optimal algorithm have total flow time plus energy $\bar{G}_o \leq 2\bar{F}_s$. Together with the bound of $\bar{F}_a$, we have $\bar{G}_a \geq F_a \geq (c \log P)\bar{F}_s \geq (c/2)(\log P)\bar{G}_o$, which completes the proof. $\square$

8 Practical Considerations

In this section, we show that our results can be applied to other models such as more general power function, discrete speed levels, and models with DVS adjustment overhead.

More general power function. So far, we assume a simple power function $P(s) = s^\alpha$. Indeed, our results can be extended for a more general power function $P(s) = c \times s^\alpha + \sigma$ where $c > 0$ and $\sigma \geq 0$ is the static power consumption. More precisely, we show that any k-competitive online algorithm for power function $P(s) = s^\alpha$ can be easily extended for the power function $P(s) = c \times s^\alpha + \sigma$, while preserving k-competitiveness for flow time plus energy.

Consider three power functions $P_1(s) = s^\alpha$, $P_2(s) = c \times s^\alpha$, and $P_3(s) = c \times s^\alpha + \sigma$. We first note that the energy consumption in the model with power function $P_2(s)$ is exactly c times the energy consumption in the model with power function $P_1(s)$. Thus to minimize the objective of total flow time plus ρ times the energy with $P_2(s)$, we can instead minimize the objective of total flow time plus $c\rho$ times the energy with $P_1(s)$, with exactly the same competitive ratio.

Then it is easy to see that any k-competitive online algorithm for $P_2(s)$ is also k-competitive for $P_3(s)$. During any time period of length ℓ, when we use $P_3(s)$ instead of $P_2(s)$, the energy usage of a schedule (both the online schedule and the optimal one) increases by an additive term $\sigma\ell$ due to the static power consumption. Thus the energy usage of the online algorithm remains at most k times the optimal schedule. Since the total flow time remains the same, the online algorithm is k-competitive for $P_3(s)$.

Discrete speed levels. Our results in previous sections assume a *continuous setting* where the processor can run arbitrarily at any speed between 0 and the maximum speed T. In the following, we consider a more realistic *discrete setting* where the processor can only run at a fixed number of discrete speed levels. Let $0 = s_0 < s_1 < \cdots < s_d = T$ be the possible speeds of the processor.

We can easily extend any algorithm A that is k-competitive for flow time plus energy to another algorithm A_d for the discrete setting and algorithm A_d is $(\Delta^\alpha k+1)$-competitive for flow time plus energy, where Δ is the maximum ratio between two consecutive non-zero speeds (e.g., if the speed levels are uniformly distributed between $[0,T]$, then $\Delta = 2$). We modify the speed functions of algorithm A to obtain algorithm A_d. Consider a particular processor. At any time t, if algorithm A runs at a speed $s(t)$ in the continuous setting, algorithm A_d runs at the lowest speed level s_i such that $s_i \geq s(t)$ in the discrete setting.

First of all, the flow time required by A_d is no more than that required by A. We then consider energy consumption. When the speed is rounded up to the next higher level, either the speed is increased by a factor of at most Δ, or the new speed is the slowest level s_1. In the former, the energy required increases by a factor of at most Δ^α. In the latter, the increase in energy consumption may be more. However, the total power consumption during the time when the latter case occurs is no larger than the total power consumption required by the optimal algorithm to run these work in the discrete setting, since the processor cannot run slower either. Summarizing the two cases, the algorithm A_d is $(\Delta^\alpha k + 1)$-competitive.

Theorem 19: Consider processors with discrete speed levels $0 = s_0 < s_1 < \cdots < s_d = T$. Let Δ be the maximum ratio between two consecutive non-zero speeds. For minimizing total flow time plus energy,

- If an algorithm A is k-competitive using processors with continuous speed in $[0,T]$, then algorithm A_d is $(\Delta^\alpha k+1)$-competitive using processors with discrete speed levels $s_0, s_1, \ldots, s_d$.
- If in the continuous setting, an algorithm A is k-competitive using processors with maximum speed $(1+\lambda)T$ for some $\lambda > 0$ (against an optimal migratory schedule with maximum speed T), then in the discrete setting, algorithm A_d is $(\max\{\Delta, (1+\lambda)\}^\alpha k + 1)$-competitive when using processors with discrete speed levels $s_0, s_1, \ldots, s_d$ and one more speed level $s_{d+1} = (1+\lambda)T$ (against an optimal migratory schedule with speed levels $s_0, s_1, \ldots, s_d$).

Models with DVS adjustment overhead. In real systems, there are overheads to perform DVS adjustments. A thorough study would also take into account such overheads. On the other hand, various experimental results, e.g., by Yuan and Qu [32], [37], showed that models with DVS adjustment overheads have very similar energy saving characteristic as an idealized model like the one we studied. They also find that the time required for a voltage (and hence speed) transition is also negligible as compared with the job execution time. We thus expect our result, which ignores the overheads, to be useful in practice.

9 Conclusions and Future Work

This paper extends the theoretical study of online scheduling for minimizing flow time plus energy to the multi-processor setting. We present and analyze a simple non-migratory online algorithm that makes use of the classified round-robin to dispatch jobs. Even in the worst case (for any job distribution), its performance is within $O(\log P)$ times of the optimal migratory offline algorithm, where P is the ratio of the maximum job size to the minimum job size. Besides theoretical work, we believe this work can also provide insights for future practical work. The next step of this work would be to conduct experimental study on our algorithm. Also, this paper focuses on sequential jobs, while the problem for parallel jobs (i.e., jobs that can run on the m processors simultaneously) can be easily reduced to the case of single-processor scheduling. Thus an interesting open problem is to consider jobs with a changing execution characteristics from fully parallelizable to completely sequential throughout their lifespans. Note that such a direction has been considered in traditional flow time scheduling [19].

Appendix

Global Critical Speed and Critical Speed

In this appendix, we prove several lemmas about global critical speed and critical speed.

Global critical speed. Albers and Fujiwara [1] observed that when scheduling a single job j on a processor for minimizing total flow time plus energy, j should be executed at the global critical speed, i.e., $1/(\alpha-1)^{1/\alpha}$.

Lemma 20: [1] At any time after a job j has been run on a processor for a while, suppose that we want to further execute j for another $x > 0$ units of work and minimize the flow time plus energy incurred to this period. The optimal strategy is to let the processor always run at the global critical speed.

Using Lemma 20, we prove Lemma 2 stated in Section 2. We restate the lemma first and then give the proof.

Lemma 2 *Given any m-processor schedule $\mathcal{S}$ for a job set J, we can construct an m-processor schedule $\mathcal{S}'$ for J such that $\mathcal{S}'$ never runs a job at speed less than the global critical speed and $\bar{G}(\mathcal{S}') \leq \bar{G}(\mathcal{S})$. Moreover, $\mathcal{S}'$ needs migration if and only if $\mathcal{S}$ does; and max-speed$(\mathcal{S}') \leq \max\{$max-speed$(\mathcal{S}), 1/(\alpha-1)^{1/\alpha}\}$.*

Proof: Assume that there is a time interval I in $\mathcal{S}$ during which a processor i is running a job j below the global critical speed. If $\mathcal{S}$ needs migration, we transform $\mathcal{S}$ to a migratory schedule $\mathcal{S}_1$ of J such that job j is always scheduled in processor i. This can be done by swapping the schedules of processor i and other processors for different time intervals. If $\mathcal{S}$ does not need migration, job j is entirely scheduled in processor i and $\mathcal{S}_1$ is simply $\mathcal{S}$. In both cases, $\bar{G}(\mathcal{S}_1) = \bar{G}(\mathcal{S})$.

We can then improve $\bar{G}(\mathcal{S}_1)$ by modifying the schedule of processor i as follows. Let x be the amount of work of j processed during I on processor i. First, we schedule this amount of work of j at the global critical speed. Note that the time required is shortened. Then we move the remaining schedule of j backward to fill up the time shortened. By Lemma 20, the flow time plus energy for j is preserved. Other jobs in J are left intact. To obtain the schedule $\mathcal{S}'$, we repeat this process to eliminate all such intervals I. □

Critical speed. We generalize the notion of global critical speed to take fractional weight into consideration. We give a non-trivial observation that when scheduling a single job j on a processor for minimizing the fractional flow time plus energy, the processor should always keep up with its critical speed $(q/(\alpha-1)p(j))^{1/\alpha}$, where $q \leq p(j)$ denotes the remaining work of job j at the time of concern. Note that the critical speed of j changes over time.

Lemma 21: At any time after a job j has been run on a processor for a while, suppose that we want to further execute j for another $x > 0$ units of work and minimize the fractional flow time plus energy incurred to this period. The optimal strategy is to let the processor always run at the critical speed.

Proof: Let $p = p(j)$, and let $q \leq p$ be the remaining work of j. We first consider the case to further process an infinitesimal amount of work (i.e., $x \to 0$). In this case we can assume that the speed s is constant. The time required is $t = x/s$, and the fractional flow time plus energy incurred, denoted by ΔG, is $s^\alpha t + (\frac{q-x/2}{p})t$. To find the optimal speed (or equivalently, optimal time) to minimize ΔG, we set $\frac{d\Delta G}{dt} = 0$. That means,

$$(1-\alpha)\left(\frac{x}{t}\right)^\alpha + \frac{q-x/2}{p} = 0 \ ,$$

or equivalently,

$$\frac{x}{t} = \left(\frac{q-x/2}{(\alpha-1)p}\right)^{1/\alpha} \quad .$$

Since $x \to 0$, we have $s = x/t = (q/(\alpha-1)p)^{1/\alpha}$.

Next, we consider x being arbitrarily large. Consider a schedule in which the processor is not running at the critical speed after processing $x' < x$ units of work. Let Δq be an infinitesimal amount. From the discussion above we can change the speed to the critical speed to obtain the optimal fractional flow time plus energy for processing the next Δq units of work. Thus we can eventually obtain a schedule that always runs at the critical speed. □

We can then prove Lemma 3 stated in Section 2.

Lemma 3 *Given any single-processor schedule $\mathcal{N}$ for a job set J, we can construct another schedule $\mathcal{N}'$ for J such that $\mathcal{N}'$ never runs a job at speed less than its critical speed and $G(\mathcal{N}') \leq G(\mathcal{N})$. Moreover, max-speed$(\mathcal{N}') \leq \max\{$max-speed$(\mathcal{N}), 1/(\alpha-1)^{1/\alpha}\}$.*

Proof: The transformation uses Lemma 21 and the same arguments in the proof of Lemma 2. □

References

[1] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM TALG*, 3(4), 2007.

[2] S. Albers, F. Muller, and S. Schmelzer. Speed scaling on parallel processors. In *Proc. SPAA*, 2007.

[3] N. Avrahami and Y. Azar. Minimizing total flow time and total completion time with immediate dispatching. In *Proc. SPAA*, pages 11–18, 2003.

[4] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. *SIAM J. on Comput*, 31(5):1370–1382, 2002.

[5] K. R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, New York, 1974.

[6] N. Bansal, H. L. Chan, T. W. Lam, and L. K. Lee. Scheduling for speed bounded processors. In *Proc. ICALP*, 2008. To appear.

[7] N. Bansal, T. Kimbrel, and K. Pruhs. Dynamic speed scaling to manage energy and temperature. *J. ACM*, 54(1), 2007.

[8] N. Bansal, K. Pruhs, and C. Stein. Speed scaling for weighted flow time. In *Proc. SODA*, pages 805–813, 2007.

[9] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[10] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[11] D. P. Bunde. Power-aware scheduling for makespan and flow. In *Proc. SPAA*, pages 190–196, 2006.

[12] H. L. Chan, W. T. Chan, T. W. Lam, L. K. Lee, K. S. Mak, and P. W. H. Wong. Energy efficient online deadline scheduling. In *Proc. SODA*, pages 795–804, 2007.

[13] H. L. Chan, T. W. Lam, and K. K. To. Nonmigratory online deadline scheduling on multiprocessors. *SIAM J. on Comput.*, 34(3):669–682, 2005.

[14] C. Chekuri, A. Goel, S. Khanna, and A. Kumar. Multi-processor scheduling to minimize flow time with ϵ resource augmentation. In *Proc. STOC*, pages 363–372, 2004.

[15] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for minimizing weighted flow time. In *Proc. STOC*, pages 84–93, 2001.

[16] Y. Chen, Z. Shao, Q. Zhuge, C. Xue, B. Xiao, and E. Sha. Minimizing Energy via Loop Scheduling and DVS for Multi-Core Embedded Systems. In *Proc. ICPADS*, pages 2–6, 2005.

[17] G. Cornuejos, G. L. Nemhauser, and L. Wosley. The uncapacitated facility location problem. In P. Mirchandani and R. Francis, editors, *Discrete Location Theory*, pages 119–171. John Wiley & Sons, 1990.

[18] D. R. Dooly, S. A. Goldman, and S. D. Scott. On-line analysis of the TCP acknowledgment delay problem. *JACM*, 48:243–273, 2001.

[19] J. Edmonds. *Scheduling in the dark*. Theoretical Computer Science, Vol. 235, No. 1, pp. 109–141, 2000.

[20] A. Fabrikant, A. Luthra, E. Maneva, C. H. Papadimitriou, and S. Shenker. On a network creation game. In *Proc. PODC*, pages 347–351, 2003.
[21] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Proc. OSDI*, pages 73–86, 2000.
[22] S. Irani and K. Pruhs. Algorithmic problems in power management. *SIGACT News*, 32(2):63–76, 2005.
[23] S. Irani, S. Shukla, and R. K. Gupta. Algorithms for power savings. *TALG*, 3(4), 2007.
[24] B. Kalyanasundaram and K. Pruhs. Eliminating migration in multi-processor scheduling. *J. Alg*, 38:2–24, 2001.
[25] A. R. Karlin, C. Kenyon, and D. Randall. Dynamic TCP acknowledgement and other stories about $e/(e-1)$. In *Proc. STOC*, pages 502–509, 2001.
[26] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proc. STOC*, pages 110–119, 1997.
[27] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.
[28] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. SOSP*, pages 89–102, 2001.
[29] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In J. Leung, editor, *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, chapter 15-1–15-41. CRC Press, 2004.
[30] K. Pruhs, P. Uthaisombut, and G. Woeginger. Getting the best response for your erg. In *Proc. SWAT*, pages 14–25, 2004.
[31] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. In *Proc. WAOA*, pages 307–319, 2005.
[32] G. Qu. What is the limit of energy saving by dynamic voltage scaling? In *Proc. IEEE/ACM ICCAD*, pages 560–563, 2001.
[33] G. Qu. Power Management of Multicore Multiple Voltage Embedded Systems by Task Scheduling. In *IEEE ICPPW*, page 34, 2007.
[34] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. OSDI*, pages 13–23, 1994.
[35] F. Xie, M. Martonosi, and S. Malik. Bounds on Power Savings Using Runtime Dynamic Voltage Scaling: An Exact Algorithm and A Linear-time Heuristic Approximation. In *Proc. ISLPED*, pages 287–292, 2005.
[36] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proc. FOCS*, pages 374–382, 1995.
[37] L. Yuan and G. Qu. Analysis of Energy Reduction on Dynamic Voltage Scaling-Enabled Systems. *IEEE TCAD*, 24(12): 1827–1837, 2005.

Tak-Wah Lam received his PhD degree in Computer Science from the University of Washington in 1988. He is now a professor in the University of Hong Kong. His research is mainly on the design and analysis of algorithms; recent interests include online scheduling, compressed indexing, and computational biology.

Lap-Kei Lee received his bachelor degree in Computer Engineering from the University of Hong Kong in 2004. He is now a PhD student in the same university. His research interests are design and analysis of algorithms, especially on online scheduling and data streams.

Isaac K. K. To received his PhD degree in Computer Science from the University of Hong Kong in 2000. He is now a post-doctoral researcher in the University of Liverpool. His interests are mainly in the theoretical study of online algorithms, especially scheduling algorithms.

Prudence W. H. Wong received her PhD degree from the University of Hong Kong in 2003. She is a Lecturer at the Department of Computer Science of the University of Liverpool. Her major research interests are on design and analysis of algorithms, especially in on-line algorithms and computational biology.