Chapter 1

Online Scheduling

Susanne Albers
University of Freiburg, Germany

1.1 Introduction

Many scheduling problems that arise in practice are inherently *online* in nature. In these settings, scheduling decisions must be made without complete information about the entire problem instance. This lack of information may stem from various sources: (1) Jobs arrive one by one as a list or even as an input stream over time. Scheduling decisions must always be made without knowledge of any future jobs. (2) The processing times of jobs are unknown initially and during run time. They become known only when jobs actually finish. (3) Machine breakdown and maintenance intervals are unknown.

Despite the handicap of not knowing the entire input we seek algorithms that compute good schedules. Obviously, computing optimal solutions is, in general, impossible and we have to resort to approximations. Today, the standard means to evaluate algorithms working online is *competitive analysis* [47]. Here an online algorithm A is compared to an optimal offline algorithm OPT that knows the entire input in advance. Consider a given scheduling problem and suppose we wish to minimize an objective function. For any input I, let $A(I)$ be the objective function value achieved by A on I and let $OPT(I)$ be the value of an optional solution for I. Online algorithm A is called *c-competitive* if there exists a constant b such that, for all problem inputs I, inequality $A(I) \leq c \cdot OPT(I) + b$ holds. The constant b must be independent of the input. An analogous definition can be set up for maximization problems. Note that competitive analysis is a strong worst-case performance measure; no probabilistic assumptions are made about the input.

Online scheduling algorithms were already investigated in the 1960s but an extensive in-depth study has only started 10 to 15 years ago, after the concept of competitive analysis had formally been introduced. By now there exists

0-8493-0052-5/00/$0.00+$.50

a rich body of literature on online scheduling. The investigated problem settings, as in standard scheduling, address various machine models (identical, related or unrelated machines), different processing formats (preemptive or non-preemptive scheduling) and various objective functions (such as makespan, (weighted) sum of completion times, (weighted) sum of flow times etc.).

In this book chapter, due to space constraints, we can only present a selection of the known results. In the first part of the chapter we focus on some classical scheduling problems and summarize the state of the art. The second part of the chapter is devoted to energy-efficient scheduling, a topic that has received quite some research interest recently and promises to be an active theme for investigation in the future. Wherever possible results and theorems are accompanied by proofs. However, for many results, the corresponding analyses are quite involved and detailed proofs are beyond the scope of the chapter.

1.2 Classical Scheduling Problems

In the first part of this section we study online algorithms for makespan minimization, which represents one of the most basic problems in scheduling theory. The second part addresses flow time based objectives. Finally we consider load balancing problems where jobs have a temporary duration. Such jobs arise e.g. in the context of telephone calls or network routing requests.

1.2.1 Makespan Minimization

Consider a basic scenario where we are given m identical machines working in parallel. As input we receive a sequence of jobs $I = J_1, J_2, \ldots, J_n$ with individual processing times, i.e. J_i has a processing time of p_i time units. The jobs arrive incrementally, one by one. Whenever a new job arrives, its processing time is known. The jobs has to be assigned immediately and irrevocably to one of the machines without knowledge of any future jobs. Preemption of jobs is not allowed. The goal is to minimize the makespan, i.e. the completion time of the last jobs that finishes in the schedule.

This fundamental scenario was investigated by Graham [27] in 1966 who devised the famous *List* scheduling algorithm. At any given time let the *load* of a machine be the sum of the processing times of the jobs currently assigned to it.

Algorithm List: Schedule any new job on the least loaded machine.

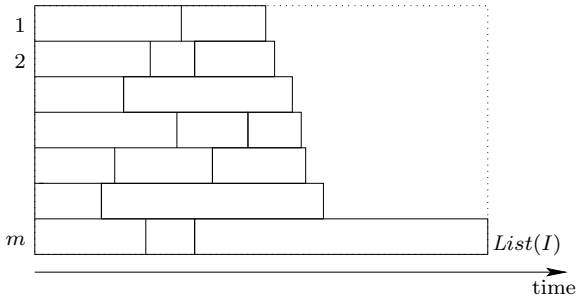

FIGURE 1.1: The schedule generated by *List*.

THEOREM 1.1 [27]
List is $(2 - \frac{1}{m})$*-competitive.*

PROOF For an arbitrary job sequence $I = J_1, \ldots, J_n$, consider the schedule constructed by *List* and let $List(I)$ be the resulting makespan. Without loss of generality we number the machines in order of non-decreasing final loads in *List*'s schedule. Then the load on machine m defines the makespan. Consider a time interval of length $List(I)$ on any of the m machines, cf. Figure 1.1. The last machine m processes jobs continuously without interruption. The first $m-1$ machines each process a subset of the jobs and then experience a (possibly empty) idle period. During their active periods the machines finish a total processing volume of $\sum_{i=1}^{n} p_i$. Consider the assignment of the last job on machine m. By the *List* scheduling rule, this machine had the smallest load at the time of the assignment. Hence, any idle period on the first $m-1$ machines cannot be longer than the processing time of the last job placed on machine m and hence cannot exceed the maximum processing time $\max_{1 \le i \le n} p_i$. We conclude

$$mList(I) \le \sum_{i=1}^{n} p_i + (m-1) \max_{1 \le i \le n} p_i,$$

which is equivalent to

$$List(I) \le \frac{1}{m} \sum_{i=1}^{n} p_i + (1 - \frac{1}{m}) \max_{1 \le i \le n} p_i.$$

Note that $\frac{1}{m} \sum_{i=1}^{n} p_i$ is a lower bound on $OPT(I)$ because the optimum makespan cannot be smaller than the average load on all the machines. Furthermore, $OPT(I) \ge \max_{1 \le i \le n} p_i$ because the largest job must be processed on some machine. We conclude $List(I) \le (2 - \frac{1}{m})OPT(I)$. ∎

Graham also showed a matching lower bound on the performance of *List*.

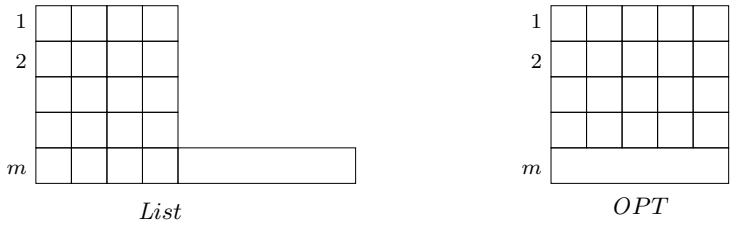

FIGURE 1.2: Schedules generated by *List* and *OPT*.

THEOREM 1.2
List does not achieve a competitive ratio smaller than $2 - \frac{1}{m}$.

PROOF Consider a job sequence I consisting of (a) $m(m-1)$ jobs, each having a processing time of 1, followed by (b) one job having a processing time of m time units. The resulting schedules generated by *List* and *OPT* are depicted in Figure 1.2. The *List* algorithm assigns the first $m(m-1)$ jobs in a *Round-Robin* fashion to machines so that a load of $m-1$ is generated. The final job of processing time m then causes a makespan of $2m-1$. On the other hand *OPT* schedules the initial small jobs on $m-1$ machines only, reserving the last machine for the final job. This gives a makespan of m, and the desired performance ratio follows. □

Faigle, Kern and Turan [22] showed that no deterministic online algorithm can have a competitive ratio smaller than $2 - \frac{1}{m}$ for $m = 2$ and $m = 3$. Thus, for these values of m, *List* is optimal. In the 1990s research focused on finding improved online algorithms for a general number m of machines. Galambos and Woeginger [24] presented an algorithm that is $(2 - \frac{1}{m} - \epsilon_m)$-competitive, where $\epsilon_m > 0$, but ϵ_m tends to 0 as m goes to infinity. The first online algorithm that achieved a competitive ratio asymptotically smaller than 2 was given by Bartal, Fiat, Karloff and Vohra [16]. Their algorithm is 1.986-competitive. The strategy was generalized by Karger, Phillips and Torng [33] who proved an upper bound of 1.945. Later, Albers presented a new algorithms that is 1.923-competitive [1]. The strategy was modified by Fleischer and Wahl who showed a bound of 1.9201, the best performance ratio known to date. We briefly describe the algorithm.

The goal of all improved algorithms, beating the bound of $2 - \frac{1}{m}$, is to maintain an *imbalanced* schedule in which some machines are lightly loaded and some are heavily loaded. In case a large job arrives, it can be assigned to a lightly loaded machine so that a makespan of $2 - \frac{1}{m}$ times the optimum value is prevented. Formally, let $c = 1 + \sqrt{(1 + \ln 2)/2}$ and, using this definition,

$$k = \lfloor \frac{2(c-1)^2 - 1}{c} m \rfloor + 1 \qquad \text{and} \qquad \alpha = \frac{2c-3}{2(c-1)}.$$

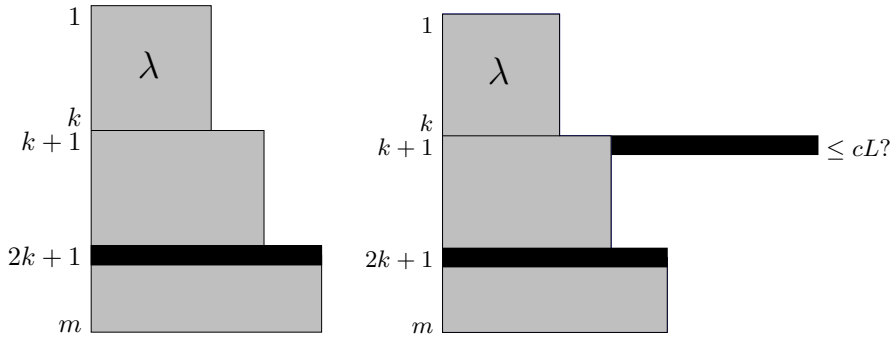

FIGURE 1.3: Schedules maintained by *Imbal*. Left part: A sample profile. Right part: The scheduling step.

We note that $k \approx \lfloor 0.36m \rfloor + 1$ and $\alpha \approx 0.46$. The algorithm by Fleischer and Wahl, called *Imbal*, tries to maintain a schedule in which k machines are lightly loaded and $m-k$ are heavily loaded. At any time we number the machines in order of non-decreasing current load, where the load of a machine is again the total processing time of jobs presently assigned to that machine. Let l_i denote the load on the i-th smallest machine, $1 \leq i \leq m$. The left part of Figure 1.3 shows a sample schedule. Moreover, let $\lambda = \frac{1}{k}\sum_{i=1}^{k} l_i$ be the average load on the k smallest machines. This average load is always compared to the load on machine $2k+1$. The goal is to maintain an *imbalanced* schedule in which $\lambda \leq \alpha l_{2k+1}$.

Each new job J_t is scheduled either on the machine with the smallest load or on the machine with the $(k+1)$-st smallest load. The decision which machine to choose depends on the current schedule. If the schedule is imbalaced, job J_t is placed on the machine with the smallest load. On the other hand, if the desired invariant $\lambda \leq \alpha l_{2k+1}$ does not hold, the algorithm considers scheduling J_t on the machine with the $(k+1)$-st smallest load, cf. the right part of Figure 1.3. The algorithm computes the resulting load $l_{k+1} + p_t$ on that machine and compares it to the average load $L = \frac{1}{m}\sum_{j=1}^{t} p_j$ on the machines after J_t is assigned. Note that L is a lower bound on the optimum makespan. If $l_{k+1} + p_t \leq cL$, then J_t is placed on machine $k+1$. Otherwise this assignment is risky and J_t is processed on the machine with the smallest load.

Algorithm Imbal: Schedule a new job J_t on the machine with the $(k+1)st$ smallest load if $\lambda > \alpha l_{2k+1}$ and $l_{k+1} + p_t \leq cL$. Otherwise schedule J_t on the machine having the smallest load.

THEOREM 1.3 [23]
Imbal achieves a competitive ratio of $1 + \sqrt{(1+\ln 2)/2} < 1.9201$.

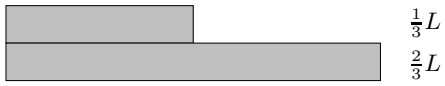

FIGURE 1.4: An ideal schedule produced by *Rand-2*.

We next turn to lower bounds for deterministic online algorithms. Recall that a result by Faigle, Kern and Turan [22] states that no deterministic strategy can be better than $(2-\frac{1}{m})$-competitive, for $m = 2$ and $m = 3$. Lower bounds for a general number m of machines were developed, for instance, in [1, 16, 22]. The best lower bound currently known is due to Rudin [44].

THEOREM 1.4 [44]
No deterministic online algorithm can achieve a competitive ratio smaller than 1.88.

An interesting open problem is to determine the exact competitiveness achievable by deterministic online strategies.

Since the publication of the paper by Bartal et al. [16], there has also been research interest in developing randomized online algorithms for makespan minimization. Bartal et al. gave a randomized algorithm for two machines that achieves an optimal competitive ratio of 4/3. This algorithm, called *Rand-2* operates as follows. For a given 2-machine schedule let the *discrepancy* be the load difference on the two machines. The algorithm tries to always maintain a schedule in which the expected discrepancy is $\frac{1}{3}L$, where $L = \sum_{j=1}^{t} p_j$ is the total processing time of jobs that have arrived so far. Figure 1.4 shows a sample schedule.

Algorithm Rand-2: Maintain a set of all schedules generated so far together with their probabilities. When a new job J_t arrives, compute the overall expected discrepancy E_1 that results if J_t were placed on the least loaded machine in each schedule. Similarly, compute the expected discrepancy E_2 if J_t were assigned to the most loaded machine in each schedule. Determine a p, $0 \leq p \leq 1$, such that $pE_1 + (1-p)E_2 \leq \frac{1}{3}L$. If such a p exists, with probability p schedule J_t on the least loaded machine and with probability $1-p$ assign it to the most loaded machine in each schedule. If such p does not exist, assign J_t to the least loaded machine.

THEOREM 1.5 [16]
Rand-2 achieves a competitive ratio of 4/3. This is optimal.

Chen et al. [21] and Sgall [46] proved that no randomized online algorithm can have a competitiveness smaller than $1/(1-(1-1/m)^m)$. This expression tends to $e/(e-1) \approx 1.58$ as $m \to \infty$. Seiden [45] presented a randomized algorithm whose competitive ratio is smaller than the best known determin-

istic ratio for $m \in \{3, \ldots, 7\}$. The competitiveness is also smaller than the deterministic lower bound for $m = 3, 4, 5$.

Recently, Albers [2] developed a randomized online algorithm that is 1.916-competitive, for all m, and hence gave the first algorithm that performs better than known deterministic algorithms for general m. She also showed that a performance guarantee of 1.916 cannot be proven for a deterministic online algorithm based on analysis techniques that have been used in the literature so far. An interesting feature of the new randomized algorithm, called *Rand*, is that at most two schedules have to be maintained at any time. In contrast, the algorithms by Bartal et al. [16] and by Seiden [45] have to maintain t schedules when t jobs have arrived. The *Rand* algorithm is a combination of two deterministic algorithms A_1 and A_2. Initially, when starting the scheduling process, *Rand* chooses A_i, $i \in \{1, 2\}$, with probability $\frac{1}{2}$ and then serves the entire job sequence using the chosen algorithm. Algorithm A_1 is a conservative strategy that tries to maintain schedules with a low makespan. On the other hand, A_2 is an aggressive strategy that aims at generating schedules with a high makespan. A challenging open problem is to design randomized online algorithms that beat the deterministic lower bound, for all m.

1.2.2 Flow Time Objectives

Minimizing the flow time of jobs is another classical objective in scheduling. In an online setting we receive a sequence $I = J_1, \ldots, J_n$ of jobs, where each job J_i is specified by an arrival time r_i and a processing time p_i. Clearly the arrival times satisfy $r_i \leq r_{i+1}$, for $1 \leq i \leq n-1$. Preemption of jobs is allowed, i.e. the processing of a job may be stopped and resumed later. We are interested in minimizing the *flow time* of jobs. The flow time of a job is the length of the time period between arrival time and completion of the job. Formally, suppose that a job released at time r_i is completed at time c_i. Then the flow time is $f_i = c_i - r_i$.

Two scenarios are of interest. In *clairvoyant scheduling*, when J_i arrives, its processing time p_i is known. This assumption is realistic in classical manufacturing or, w.r.t. to new application areas, in the context of a web server delivering static web pages. In *non-clairvoyant scheduling*, when J_i arrives, p_i is unknown and becomes known only when the job finishes. This assumption is realistic in operating systems.

We first study clairvoyant scheduling and focus on the objective of minimizing the sum of flow times $\sum_{i=1}^{n} f_i$. The most classical scheduling algorithm is *Shortest Remaining Processing Time*.

Algorithm Shortest Remaining Processing Time (SRPT): At any time execute the job with the least remaining work.

It is well known and easy to verify that *SRPT* constructs optimal schedules on one machine, see e.g. [12].

THEOREM 1.6
For one machine, SRPT is 1-competitive.

For m machines, *SRPT* also achieves the best possible performance ratio, but the analysis is considerably more involved. A first, very sophisticated analysis was given by Leonardi and Raz [36]. A simplified proof was later presented by Leonardi [35]. In the following let $p_{\min} = \min_{1 \le i \le n} p_i$ and $p_{\max} = \max_{1 \le i \le n} p_i$ be the smallest and largest processing times, respectively. Moreover, $P = p_{\max}/p_{\min}$.

THEOREM 1.7 [37]
For m machines, SRPT has a competitive ratio of $O(\min\{\log P, \log(n/m)\})$.

The above competitiveness is best possible.

THEOREM 1.8 [37]
For m machines, any randomized online algorithms has a competitive ratio of $\Omega(\log(n/m))$ *and* $\Omega(\log P)$.

While *SRPT* is a classical algorithm and achieves an optimal performance ratio, it uses *migration*, i.e. a job when being preempted may be moved to another machine. In many practical settings, this is undesirable as the incurred overhead is large. Awerbuch et al. [7] gave a refined algorithm that does not use migration and is $O(\min\{\log P, \log n\})$-competitive. Chekuri et al. [20] gave a non-preemptive algorithm achieving an optimal performance of $O(\min\{\log P, \log(n/m)\})$.

The algorithms mentioned above have optimal competitive ratios; however the performance guarantees are not constant. Interestingly, it is possible to improve the bounds using *resource augmentation*, i.e. an algorithm is given processors of higher speed. Here it is assumed that an optimal offline algorithms operates with machines of speed 1, while an online strategy may use machines running at speed $s \ge 1$. In this context, the best result is due to McCullough and Torng [39] who showed that, using speed $s \ge 2 - 1/m$, *SRPT* is $(1/s)$-competitive.

An interesting, relatively new performance measure is the *stretch* of a job, which is defined as the flow time divided by the processing time, i.e. $st_i = f_i/p_i$. The motivation for this metric is that a user is willing to wait longer for the completion of long jobs; a quick response is expected for short jobs. Consider the objective of minimizing the total stretch $\sum_{i=1}^{n} f_i/p_i$ of jobs. Muthukrishan et al. [41] showed that *SRPT* performs very well.

THEOREM 1.9 [41]
On one machine SRPT is 2-competitive.

Muthukrishan et al. [41] established an almost matching lower bound on the performance of *SRPT* and proved that no online algorithm can be better than 1.036-competitive. We next turn to parallel machines.

THEOREM 1.10 [41]
For m machines, SRPT is 14-competitive.

Chekuri et al. [20] developed an improved 9.82-competitive algorithm and presented a 17.32-competitive strategy not using any migration.

We next address non-clairvoyant scheduling where the processing time of an incoming job is not known in advance. We focus again on the objective of minimizing the total flow time $\sum_{i=1}^{n} f_i$ of jobs and concentrate on one machine. In this setting a natural algorithm is *Round Robin*, which always assigns an equal amoung of processing resources to all the jobs. Kalyanasundaram and Pruhs [31] showed that the algorithms does not perform well relative to the optimum. The competitiveness is at least $\Omega(n/\log n)$. In fact any deterministic algorithm does not perform very well.

THEOREM 1.11 [40]
Any deterministic online algorithm has a competitive ratio of $\Omega(n^{1/3})$.

Again, resource augmentation proves to be a very powerful tool in this context. Kalyanasundaram and Pruhs [31] analyzed the following algorithm.

Algorithm Shortest Elapsed Time First (SETF): At any time execute the job that has been processed the least.

THEOREM 1.12 [31]
For any $\epsilon > 0$, using a speed of $1+\epsilon$, SETF achieves a competitive ratio of $1 + 1/\epsilon$.

Finally, using randomization it is possible to get down to a logarithmic bound of $\Theta(\log n)$ without resource augmentation, see [17] and [32].

1.2.3 Load Balancing

In this section we study load balancing problems that arise in new applications. Consider, for instance, a set of satellite links on which phone calls have to be scheduled or, alternatively, a set of network links on which data transmission requests have to served. In these scenarios the requests have an unknown duration and incur a certain load or congestion on the chosen link. The goal is to minimize the total load that ever occurs on any of the links. These problem settings can be formalized as follows.

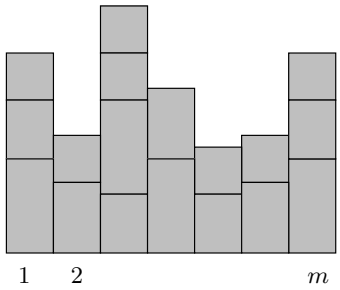

FIGURE 1.5: A load profile with m machines.

We are given a set of m machines (representing the available links). A sequence of jobs $I = J_1, J_2, \ldots, J_n$ arrives online. Let r_i denote the arrival time of J_i. Job J_i has an unknown duration and incurs a load of l_i when assigned to a machine. For any time t, let $L_j(t)$ denote the load of machine j, $1 \leq i \leq m$, which is the sum of the loads of the jobs present on machine j at time t. The goal is to minimize the maximum load that occurs during the processing of I. Figure 1.5 depicts a sample schedule with m machines. The rectangles represent jobs, where the height of a rectangle corresponds to the load of the job.

We concentrate on settings with m identical machines. A natural online strategy is *Greedy* which assigns an incoming job to a machine currently having the smallest load. Azar and Epstein [9] observed that *Greedy* is $(2 - \frac{1}{m})$-competitive. They also proved that this is the best competitiveness achievable by a deterministic strategy. In the following we will study the scenario with identical machines and *restricted assignment*, i.e. each job J_i can only be assigned to a subset M_i of admissible machines. Azar et al. [8] showed that *Greedy* is $\Theta(m^{2/3})$-competitive. They also proved that no online algorithm can achieve a competitiveness smaller than $\Omega(\sqrt{m})$. Azar et al. [11] gave a matching upper bound. The corresponding algorithm is called *Robin Hood*. Since both the algorithm and its analysis are simple and elegant, we give the details.

At any time *Robin Hood* maintains a lower bound B on the optimum load $OPT(I)$ incurred on the given job sequence I. Initially, let $B(0) = 0$. At any time t, this bound is updated as follows. If no new job arrives at thime t, then $B(t) = B(t-1)$. If a new job J_i arrives at time $t = r_i$, then the update is as follows. Again, let $L_j(t-1)$ denote the load on machine j at the end of time step $t-1$.

$$B(t) := \max\{B(t-1), l_i, \frac{1}{m}(l_i + \sum_{j=1}^{m} L_j(t-1))\}.$$

Clearly, $B(t) \leq OPT(I)$ always as we are interested in the maximum load that ever occurs on the machines. At any time t a machine is called *rich* if its

current load is at least $\sqrt{m}B(t)$; otherwise the machine is called *poor*.

Algorithm Robin Hood: When a new job J_i arrives, if possible assign it to a poor machine. Otherwise assign it to the rich machine that became rich most recently.

THEOREM 1.13 [11]
Robin Hood is $O(\sqrt{m})$-competitive.

PROOF We start with a simple lemma estimating the number of rich machines.

LEMMA 1.1
There always exist at most $\sqrt{m}$ machines that are rich.

PROOF The number of rich machines can only increase when new jobs are assigned to machines. So consider any time $t = r_i$. If more than $\sqrt{m}$ machines were rich after the assignment of J_i, then the aggregate load on the m machines would be greater than $\sqrt{m}\sqrt{m}B(t) = mB(t)$. However, by the definition of $B(t)$, $B(t) \geq \frac{1}{m}(l_i + \sum_{j=1}^{m} L_j(t-1))$, i.e. $mB(t)$ is an upper bound on the aggregate load. □

In order to establish the theorem we prove that whenever *Robin Hood* assigns a job J_i to a machine j, the resulting load on the machine is upper bounded by $(2\sqrt{m}+1)OPT(I)$.

First suppose that j is poor. Then the new load on the machine is $L_j(r_i - 1) + l_i < \sqrt{m}B(r_i) + B(r_i) \leq (\sqrt{m}+1)OPT(I)$.

Next suppose that machine j is rich when J_i is assigned. Let $r_{t(i)}$ be the most recent point in time when machine j became rich. Furthermore, let S be the set of jobs that are assigned to machine j in the interval $(r_{t(i)}, r_i]$. Any job $J_k \in S$ could only be assigned to machines that were rich at time $r_{t(i)}$ because *Robin Hood* places a job on the machine that became rich most recently if no poor machines are available. Let

$$h = |\bigcup_{J_k \in S} M_k|.$$

Then, by the Lemma 1.1 $h \leq \sqrt{m}$. Since only $\sqrt{m}$ machines are available for jobs in S we have $OPT(I) \geq \frac{1}{\sqrt{m}} \sum_{J_k \in S} l_k$ and hence

$$\sum_{J_k \in S} l_k \leq \sqrt{m}OPT(I).$$

Hence the resulting load on machine j after the assignment of J_i is at most

$$L_j(r_{t(i)} - 1) + l_{t(i)} + \sum_{J_k \in S} l_k < \sqrt{m}B(r_{t(i)}) + l_{t(i)} + \sqrt{m}OPT(I)$$
$$\leq (2\sqrt{m} + 1)OPT(I).$$

This concludes the proof. □

Further work on online load balancing can be found, e.g., in [5, 10].

1.3 Energy-Efficient Scheduling

In many computational environments energy has become a scare and/or expensive resource. Consider, for instance, battery-operated devices such as laptops or mobile phones. Here the amount of available energy is severely limited. By performing tasks with low total energy, one can considerably extend the lifetime of a given device. Generally speaking, the energy consumption in computer systems has grown exponentially over the past years. This increase is strongly related to Moore's law which states that the number of transistors that can be placed on an integrated circuit doubles approximately every two years. Since transistors consume energy, increased transistor density leads to increased energy consumption. Moreover, electricity costs impose a substantial strain on the budget of data and computing centers, where servers and, in particular, CPUs account for 50–60% of the energy consumption. In fact, Google engineers, maintaining thousands of servers, recently warned that if power consumption continues to grow, power costs can easily overtake hardware costs by a large margin [15]. Finally, a high energy consumption is critical because most of the consumed energy is eventually converted into heat which can harm the electronic components of a system.

In this section we study algorithmic techniques to save energy. It turns out that these techniques are actually scheduling strategies. There exist basically two approaches.

- **Power-down mechanisms:** When a system is idle, move it into lower power stand-by or sleep modes. We are interested in scheduling algorithms that perform the transitions at the "right" points in time.
- **Dynamic speed scaling:** Modern microprocessors, such as Intel XScale, Intel Speed Step or AMD Power Now, can run at variable speed. The higher the speed, the higher the power consumption. We seek algorithms that use the speed spectrum in an optimal way.

Obviously, the goal of both the above techniques is to minimize the consumed energy. However, this has to be done subject to certain constraints, i.e. we

have to design feasible schedules or must provide a certain quality of service to a user.

Over the last years there has been considerable research interest in scheduling strategies saving energy. In the following we first address power-down mechanisms and then study dynamic speed scaling.

1.3.1 Power-Down Mechanisms

We start by analyzing a simple 2-state system. Consider a processor or machine that can reside in one of two possible states. Firstly, there is an *active state* that consumes one energy unit per time unit. Secondly, there exists a *sleep state* consuming zero energy units per time unit. Transitioning the machine from the active to the sleep state and, at some later point, back to the active state requires D energy units, where $D \geq 1$. As input we receive an alternating sequence of active and idle time periods. During each active period the machine has to be in the active mode. During any idle period the machine has the option of powering down to the sleep state. The goal is to find a schedule, specifying the (possible) power-down time for each idle interval, that minimizes the total energy consumption.

Since the energy consumption is fixed in the active periods, optimization strategies focus on the idle periods. In the offline scenario, the length of any idle period is known in advance. In the online setting the length of an idle period is not known in advance and becomes known only when the period ends. We remark that we ignore the latency incurred by a power-up operation to the active mode; we focus on the mere energy consumption. In the following we present online algorithms A that, for any idle time period I, incur an energy consumption that is at most c times the optimum consumption in I, for some $c \geq 1$. Obviously, such an algorithm A is c-competitive, for any alternating sequence of active and idle time periods. We note that the problem of minimizing energy consumption in any idle period I is equivalent to the famous ski rental problem [29].

In the following we focus on one particular idle time period I. An optimal offline algorithm is easy to state. If the length of I is larger D time units, power down immediately at the beginning of I. Otherwise reside in the active mode throughout I. Next we present an optimal online algorithm.

Algorithm Alg-2: Power down to the sleep mode after D time units if the idle period I has not ended yet.

THEOREM 1.14
Alg-2 is 2-competitive.

PROOF Let T denote the length of I. If $T < D$, then *Alg-2* does not power down in I and incurs an energy consumption of T, which is equal to the

consumption of an optimal offline algorithm OPT. On the other hand, if $T \geq D$, *Alg-2* powers down after D time units and the total energy consumption is $D + D = 2D$. In this case OPT pays a cost of D, and the desired performance ratio follows. □

THEOREM 1.15
No deterministic online algorithm A can achieve a competitive ratio smaller than 2.

PROOF An adversary observes the behavior of A during an idle time period I. As soon as A powers down to the sleep state, the adversary terminates I. The energy consumption of A is $T + D$, where $T = |I|$ is the length of the idle period. The theorem now follows because an optimal offline algorithm pays $\min\{T, D\}$. □

Using randomization one can improve the competitive ratio.

Algorithm RAlg-2: In an idle period power down to the sleep mode after t time units according to the probability density function p_t, where

$$p_t = \begin{cases} \frac{1}{(e-1)D} e^{t/D} & 0 \leq t \leq D \\ 0 & \text{otherwise} \end{cases}$$

THEOREM 1.16 [34]
RAlg-2 achieves a competitive ratio of $e/(e-1) \approx 1.58$.

The above performance guarantee is best possible.

THEOREM 1.17 [34]
No randomized online algorithm achieves a competitive ratio smaller than $e/(e-1) \approx 1.58$.

In practice, rather than in worst-case analysis, one might be interested in a probabilistic setting where the length of idle time periods is governed by a probabiliy distribution. Let $Q = (q_T)_{0 \leq T < \infty}$ be a probability distribution on the length of idle periods. Let A_t be the deterministic algorithm that powers down after t time units. The expected energy consumption of A_t on idle periods whose length is generated according to Q is

$$E[A_t(I_Q)] = \int_0^t T q_T dT + (t + D) \int_t^\infty q_T dT.$$

Let A_Q^* be the algorithm A_t that minimizes the above expression. This algorithm performs well relative to the expected optimum cost $E[OPT(I_Q)]$.

THEOREM 1.18 [34]
For any probability distribution Q, the best corresponding algorithm A_Q^* satisfies

$$E[A_Q^*(I_Q)] \leq \frac{e}{(e-1)} E[OPT(I_Q)].$$

Irani et al. [28] and Augustine et al. [6] extended many of the above results to systems/machines that can reside in several states. A specification of such systems is given, for instance, in the Advanced Configuration and Power Management Interface (ACPI), which establishes industry-standard interfaces enabling power management and thermal management of mobile, desktop and server platforms.

In general, consider a system/machine consisting of $l+1$ states $s_0, \ldots, s_l$, where s_0 is the active mode. Let r_i denote the energy consumption per time unit in state s_i, $0 \leq i \leq l$. These rates satisfy $r_i > r_{i+1}$, for $0 \leq i \leq l-1$. Furthermore, let d_{ij} denote the cost of transitioning from state s_i to s_j. We assume that the triangle inequality $d_{ij} \leq d_{ik} + d_{kj}$ holds for any i, j and k.

Augustine et al. [6] first argue that if the machine powers up, then it powers up to the active mode. Furthermore, we can assume without loss of generality that the power-up cost to the active state is zero, i.e. $d_{i0} = 0$, for any $1 \leq i \leq l$. If $d_{i0} > 0$, for some i, then we can define a new system with $d'_{ij} = d_{ij} + d_{j0} - d_{i0}$ for $i < j$ and $d_{ij} = 0$, for $j < i$. The total transition cost in the new system is exactly equal to the cost in the original one.

Let $D(i) = d_{0i}$ be the power-down cost into state i. Then the energy consumption of an optimal offline algorithm OPT in an idle period of length t is

$$OPT(t) = \min_{0 \leq i \leq l} \{D(i) + r_i t\}.$$

Interestingly, the optimal cost has a simple graphical representation, cf. Figure 1.6. If we consider all linear functions $f_i(t) = D(i) + r_i t$, then the optimum energy consumption is given by the lower envelope of the arrangement of lines. We can use this lower envelope to guide an online algorithm which state to use at any time. Let $S(t)$ denote the state used by OPT in an idle period of length t, i.e. $S(t)$ is the state $\arg\min_{0 \leq i \leq l}\{D(i) + r_i t\}$. The following algorithm traverses the state sequence as suggested by the optimum offline algorithm.

Algorithm Lower Envelope (LE): In an idle period, at any time t, use state $S(t)$.

Irani et al. [28] analyzed the above algorithm in *additive systems*, where for any states $i < k < j$ we have $d_{ij} = d_{ik} + d_{kj}$.

THEOREM 1.19 [28]
Algorithm LE is 2-competitive.

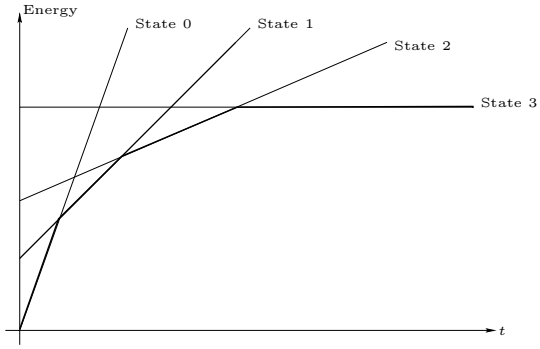

FIGURE 1.6: The optimum solution in a multi-state system.

Algorithm *LE* can be extended to work in non-additive systems where, in general, the triangle inequality $d_{ij} \leq d_{ik} + d_{kj}$ holds for any $i < k < j$. Let $\mathcal{S} = \{s_0, \ldots, s_l\}$ be the original set of states. We first construct a state set $\mathcal{S}' \subseteq \mathcal{S}$ such that for any states $s_i, s_j \in \mathcal{S}'$, with $i < j$, relation $\gamma D(i) \leq D(j)$ is satisfied, where $\gamma = 1 + 1/\sqrt{2}$. Such a set is easy to construct. Initially, let $\mathcal{S}' = \{s_l\}$. We then traverse the original states in order of decreasing index. Let s_j be the last state added to $\mathcal{S}'$. We now determine the largest i, where $i < j$, such that $\gamma D(i) \leq D(j)$. This state s_i is next added to $\mathcal{S}'$. At the end of the construction we have $s_0 \in \mathcal{S}'$ because $D(0) = 0$. Let OPT' denote the optimum offline algorithm using state set $\mathcal{S}'$ and let $S'(t)$ be the state used by OPT' in an idle period of length t.

Algorithm LE': In an idle period, at any time t, use state $S'(t)$.

THEOREM 1.20 [6]

Algorithm LE' achieves a competitive ratio of $3 + 2\sqrt{2} \approx 5.8$.

The above theorem ensures a competitiveness of $3 + 2\sqrt{2}$ in *any* multi-state system. Better performance ratios are possible for specific systems. Augustine et al. [6] developed an algorithm that achieves a competitive ratio of $c^* + \epsilon$, for any $\epsilon > 0$, where c^* is the best competitive ratio achievable for a given system. The main idea of the solution is to construct an efficient strategy A that decides, for a given c, if a c-competitive algorithm exists for the given architecture. The best value of c can then be determined using binary search in the interval $[1, 3 + 2\sqrt{2}]$.

THEOREM 1.21 [6]

A $(c^*+\epsilon)$*-competitive online algorithm can be constructed in* $O(l^2 \log l \log(1/\epsilon))$ *time.*

Irani et al. [28] and Augustine et al. [6] also present various results for scenarios where the length of idle periods is governed by probability distributions. Furthermore, the first paper [28] contains an interesting experimental study on an IBM Hard Drive with four states.

1.3.2 Dynamic Speed Scaling

In this section we study the problem of dynamically adjusting the speed of a variable-speed processor/machine so as to minimize the total energy consumption. Consider a machine that can run at variable speed s. The higher the speed, the higher the energy consumption is. More formally, at speed s, the energy consumption is $E(s) = s^\alpha$ per time unit, where $\alpha > 1$ is a constant. In practical applications, α is typically in the range $[2, 3]$.

Over the past years, dynamic speed scaling has received considerable research interest and several scheduling problems have been investigated. However, most of the previous work focuses on deadline-based scheduling, a scenario considered in a seminal paper by Yao et al. [48]. In this setting we are given a sequence $I = J_1, \ldots, J_n$ of jobs. Job J_i is released at time r_i and must be finished by a deadline d_i. We assume $r_i \leq r_{i+1}$, for $1 \leq i < n$. To finish J_i a processing volume of p_i must be completed. This processing volume, intuitively, can be viewed as the number of CPU cycles necessary to complete the job. The time it takes to finish the job depends on the processor speed. Using, for instance, a constant speed s, the execution time is p_i/s. Of course, over time, a variable speed may be used. Preemption of jobs is allowed. The goal is to construct a feasible schedule, observing the release times and deadlines, that minimizes the total energy consumption.

The paper by Yao et al. [48] assumes that (a) the machine can run at a continuous spectrum of speeds and (b) there is no upper bound on the maximum speed. Condition (b) ensures that there is always a feasible schedule. Later, we will discuss how to remove these constraints. For the above deadline-based scheduling problem, again, two scenarios are of interest. In the offline setting, all jobs of I along with their characteristics are completely known in advance. In the online variant of the problem, the jobs arrive over time. At any time future jobs are unknown. It turns out that the offline problem is interesting in itself and can be used to design good online strategies. For this reason, we first address the offline problem and present an algorithm proposed by Yao, Demers and Shenker [48].

The strategy is known as the *YDS* algorithm, referring to the initials of the inventors, and computes the *density* of time intervals. Given a time interval $I = [t, t']$, the density is defined as

$$\Delta_I = \frac{1}{|I|} \sum_{[r_i, d_i] \subseteq I} p_i.$$

Intuitively, Δ_I is the minimum average speed necessary to complete all jobs

that must be scheduled in I. Let S_I be the set of jobs J_i hat must be processed in I, i.e. that satisfy $[r_i, d_i] \subseteq I$. Algorithm *YDS* repeatedly determines the interval I of maximum density. In I it schedules the jobs of S_I at speed Δ_I using the *Earliest Deadline First* policy. Then set S_I as well as time interval I are removed from the problem instance.

Algorithm YDS: Initially, let $\mathcal{J} = \{J_1, \ldots, J_n\}$. While $\mathcal{J}$ is not empty, execute the following two steps. (1) Determine the time interval $I = [t, t']$ of maximum density Δ_I along with the job set S_I. In I process the jobs of S_I at speed Δ_I according to the *Earliest Deadline First* policy. (2) Reduce the problem instance by I. More specifically, set $\mathcal{J} := \mathcal{J} \setminus S_I$. For any $J_i \in \mathcal{J}$ with $r_i \in I$, set $r_i := t'$. For any $J_i \in \mathcal{J}$ with $d_i \in I$, set $d_i := t$. Remove I from the time horizon.

Obviously, when identifying intervals of maximum density, it suffices to consider $I = [t, t']$ for which the interval boundaries are equal to the release times r_i and deadlines d_i of the jobs.

THEOREM 1.22 [48]
Algorithm YDS constructs a feasible schedule that minimizes the total energy consumption.

Feasibility of the constructed schedule follows from the fact that, for each interval I of maximum density identified in the various iterations of *YDS*, the algorithm constructs a feasible schedule in I. Optimality follows from the convexity of the energy consumption function $E(s) = s^\alpha$: Suppose that the machine runs at speed s_1 for ϵ time units and at speed s_2 for another ϵ time units. Assume $s_1 < s_2$. Then a schedule with a strictly smaller energy consumption can be achieved by using speed $(s_1 + s_2)/2$ in both periods of length ϵ. This holds because $\epsilon s_1^\alpha + \epsilon s_2^\alpha > 2\epsilon(\frac{s_1+s_2}{2})^\alpha$ is equivalent to $\frac{1}{2}(s_1^\alpha + s_2^\alpha) > (\frac{s_1+s_2}{2})^\alpha$, and the latter inequality can easily be verified using Figure 1.7. This convexity argument implies that it is not reasonable to vary the speed in an interval I of maximum density. Moreover, it is not reasonable to increase the speed in I while reducing the speed outside I. Optimality of the resulting schedule then follows.

A straightforward implementation of *YDS* runs in time $O(n^3)$. Gaujal et al. [26] and Gaujal and Navet [25] gave algorithms achieving improved running times for some specific classes of input instances.

Algorithm *YDS* assumes a continuous spectrum of speeds. In practice only a finite set of speed levels $s_1 < s_2 < \ldots < s_d$ is available. *YDS* can be adapted easily for feasible job instances, i.e. a feasible schedule exists for the available set of speeds. Obviously, feasibility can be checked easily by always using the maximum speed s_d and scheduling the available jobs according to the *Earliest Deadline First* policy. Given a feasible job instance, we first construct the schedule according to *YDS*. For each identified interval I of

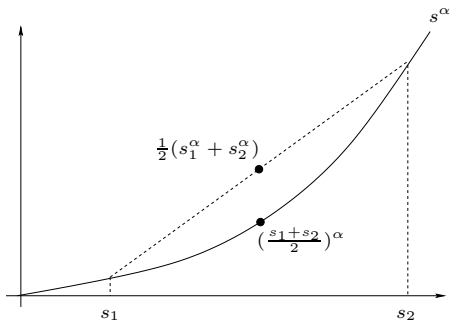

FIGURE 1.7: The convexity of the energy consumption function.

maximum density we approximate the desired speed Δ_I by the two adjacent speed levels $s_{k+1} > \Delta_I > s_k$. Speed s_{k+1} is used for the first δ time units and s_k is used for the last $|I| - \delta$ in I time units, where δ is chosen such that $\delta s_{k+1} + (|I| - \delta)s_k = |I|\Delta_I$. Here $|I|$ denotes the length of I. An algorithm with an improved running time of $O(dn \log n)$ was presented by Li and Yao [38].

We next turn to online algorithms and consider again a continuous unbounded spectrum of speeds. We assume that whenever a new job J_i arrives at time r_i, its deadline d_i and processing volume p_i are known. Yao et al. [48] devised two elegant online strategies called *Average Rate* and *Optimal Available*. For any incoming job J_i, *Average Rate* considers the *density* $\delta_i = p_i/(d_i - r_i)$, which is the minimum average speed necessary to complete the job in time if no other jobs were present. At any time t the speed $s(t)$ is set to the accumulated density of unfinished jobs present at time t.

Algorithm Average Rate: At any time t use a speed of $s(t) = \sum_{J_i : t \in [r_i, d_i]} \delta_i$. Among the available unfinished jobs, process the one whose deadline is earliest in the future.

THEOREM 1.23 [48]

For any $\alpha \geq 2$, the competitive ratio c of Average Rate satisfies $\alpha^\alpha \leq c \leq 2^{\alpha-1}\alpha^\alpha$.

The strategy *Optimal Available* is computationally more expensive in that it always computes an optimal schedule for the currently available work load. This can be done using algorithm *YDS*.

Algorithm Optimal Available: Whenever a new job arrives, compute an optimal schedule for the currently available, unfinished jobs.

Bansal et al. [13] gave a comprehensive analysis of the above algorithm. It shows that *Optimal Available* is at least as good as *Average Rate*.

THEOREM 1.24 [13]
Optimal Available achieves a competitive ratio of exactly α^α.

Bansal et al. [13] also presented a new online algorithm that tries to approximate the optimal speeds of *YDS* more closely. For times t, t_1 and t_2, let $w(t, t_1, t_2)$ be the total processing volume of jobs that have arrived by time t have a release time of at least t_1 and a deadline of at most t_2.

Algorithm BKP: At any time t use a speed of

$$s(t) = \max_{t'>t} \frac{w(t, et - (e-1)t', t')}{t' - t}.$$

Always process the available unfinished job whose deadline is earliest in the future.

THEOREM 1.25 [13]
Algorithm BKP achieves a competitive ratio of $2(\frac{\alpha}{\alpha-1})^\alpha e^\alpha$.

The competitiveness of all online algorithms presented so far depends exponentially on α. Bansal et al. [13] demonstrated that this exponential dependence is inherent to the problem.

THEOREM 1.26 [13]
The competitive ratio of any randomized online algorithm is at least $\Omega((\frac{4}{3})^\alpha)$.

An interesting open problem is to settle the exact competitiveness that can be achieved by online strategies.

Chan et al. [19] present an online algorithm for the scenario that there exists a maximum speed $s_{\max}$; below this upper bound the spectrum of speeds is still continuous. The algorithm has the interesting feature that it can handle infeasible job instances by discarding jobs if the work load becomes too high. The proposed strategy achieves a constant factor in terms of throughput and in terms of energy. The throughput is the total processing volume of successfully completed jobs. Since we aim at throughput maximization, a strategy A is c-competitive w.r.t. this performance measure if the throughput A is at least $1/c$ times the optimum throughput.

Whenever a new job J_i arrives, the algorithm proposed by Chan et al. [19] checks if the job should be admitted. In this context, a set S of jobs is called *full-speed admissible* if all jobs in S can be completed by their deadline using a maximum speed of $s_{\max}$. Speed values are used according to *Optimal Available*.

Algorithm FSA(OAT): At any time maintain a set S of admitted jobs. Number the jobs $J_{i_1}, \ldots, J_{i_k}$ in S in order of non-decreasing deadlines. A new job J_i is admitted to S if (a) $S \cup \{J_i\}$ is full-speed admissible or if

(b) $p_i > 2(p_{i_1} + \ldots + p_{i_l})$ and $\{J_i, J_{i_{l+1}}, \ldots, J_{i_k}\}$ is full-speed admissible, where $l \in [1, k]$ is the smallest integer satisfying this constraint. In the latter case $\{J_{i_1}, \ldots, J_{i_l}\}$ are removed from S. Whenever a job is finished or its deadline has passed, it is discarded from S. At any time t use the speed that *Optimal Available* would use for the workload in S, provided that this speed is not larger then $s_{\max}$. Otherwise use $s_{\max}$.

THEOREM 1.27 [19]
FSA(OAT) achieves a competitive ratio of 14 in terms of throughput and a competitive ratio of $\alpha^\alpha + \alpha^2 4^\alpha$ in terms of energy.

If there is no upper bound on the speed, *FSA(OAT)* mimicks *Optimal Available*. Hence it achieves an optimal throughput and a competitive ratio of α^α in terms of energy consumption.

So far in this section we have studied single-machine architectures. However, power consumption is also a major concern in multi-processor environments. Modern server systems are usually equipped with several CPUs. Furthermore, many laptops today feature a dual-processor architecture and the chip manufacturer AMD has even announced a "quad-core design". In the following we investigate the problem of minimizing energy consumption in parallel machine environments. We assume that we are given m identical variable-speed machines working in parallel. As before, we consider deadline-based scheduling where a sequence $I = J_1, \ldots, J_n$, each specified by a release time, a deadline and a processing volume must be scheduled. Preemption of jobs is allowed. However migration of job is disallowed, i.e. whenever a job is preempted, it may not be move to another machine as such an operation incurs considerable overhead in practice. The goal is to minimize the total energy consumed on all the m machines.

Albers et al. [4] present a comprehensive study of the problem. They first consider the offline scenario and settle the complexity of settings with unit-size jobs, i.e. $p_i = 1$ for all $1 \leq i \leq n$. They show that problem instances with *agreeable deadlines* are polynomially solvable while instances with arbitrary deadlines are NP-hard. In practice, problem instances with agreeable deadlines form a natural input class where, intuitively, jobs arriving at later times may be finished later. Formally, deadlines are agreeable if, for any two jobs J_i and $J_{i'}$, relation $r_i < r_{i'}$ implies $d_i \leq d_{i'}$. We briefly describe the proposed algorithm as it is an interesting application of the *Round Robin* strategy.

Algorithm RR: Given a sequence of jobs with agreeable deadlines, execute the following two steps. (1) Number the jobs in order of non-decreasing release dates. Jobs having the same release date are numbered in order of non-decreasing deadlines. Ties may be broken arbitrarily. (2) Given the sorted list of jobs computed in step (1), assign the jobs to machines using the Round Robin policy. For each machine, given the jobs assigned to it, compute an optimal service schedule, using e.g. *YDS*.

THEOREM 1.28 [4]
For a set of unit size jobs with agreeable deadlines, algorithm RR computes an optimal schedule.

Paper [4] also develops various polynomial time approximation algorithms for both unit-size and arbitrary-size jobs. Again, let $\delta_i = p_i/(d_i - r_i)$ be the density of job J_i. The next algorithm partitions jobs into classes such that, within each class, job densities differ by a factor of at most 2. Formally, let $\Delta = \max_{1 \leq i \leq n} \delta_i$ be the maximum job density. Partition jobs $J_1, \ldots, J_n$ into classes C_k, $k \geq 0$, such that class C_0 contains all jobs of density Δ and C_k, $k \geq 1$, contains all jobs i with density $\delta_i \in [\Delta 2^{-k}, \Delta 2^{-(k-1)})$.

Algorithm Classified Round Robin (CRR): Execute the following two steps. (1) For each class C_k, first sort the jobs in non-decreasing order of release dates. Jobs having the same release date are sorted in order of non-decreasing deadline. Then assign the jobs of C_k to processors according to the *Round Robin* policy, ignoring job assignments done for other classes. (2) For each processor, given the jobs assigned to it, compute an optimal service schedule.

THEOREM 1.29 [4]
CRR achieves an approximation factor of $\alpha^\alpha 2^{4\alpha}$ for problem instances consisting of (a) unit size jobs with arbitrary deadlines or (b) arbitrary size jobs with agreeable deadlines.

Albers et al. [4] show that improved approximation guarantees can be achieved for problem settings where all jobs have a common release time or, symmetrically, have a common deadline. Furthermore, the authors give various competitive online algorithms.

In this section we have focused on speed scaling algorithms for deadline based scheduling problems. The literature also contains results on other objectives. References [3, 14, 43] consider the minimization of flow time objectives while keeping energy consumption low. Bunde additionally investigates makespan minimization [18].

1.4 Conclusions

In this book chapter we have surveyed important results in the area of online scheduling, addressing both classical results and contributions that were developed in the past few years. The survey is by no means exhaustive. As for results on classical scheduling problems, an extensive survey article was

written by Pruhs et al. [42] and is part of a comprehensive handbook on scheduling. The most promising and fruitful direction for future research is the field of energy-efficient scheduling. In this chapter we have presented some basic results. Another survey summarizing the state of the art was presented by Irani and Pruhs [30] in 2005. The field of energy-efficient scheduling is extremely active and new results are being published in the ongoing conferences. There is a host of open problems that deserves investigation.

References

[1] S. Albers. Better bounds for online scheduling. *SIAM Journal on Computing*, 29:459–473, 1999.

[2] S. Albers. On randomized online scheduling. *Proc. 34th ACM Symposium on Theory of Computing*, 134–143, 2002.

[3] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. *Proc. 23rd International Symposium on Theoretical Aspects of Computer Science (STACS)*, Springer LNCS 3884, 621-633, 2006.

[4] S. Albers, F. Müller and S. Schmelzer. Speed scaling on parallel processors. *Proc. 19th ACM Symposium on Parallelism in Algorithms and Architectures*, 289–298, 2007.

[5] A. Armon, Y. Azar and L. Epstein. Temporary tasks assignment resolved. *Algorithmica*, 36(3):295–314, 2003.

[6] J. Augustine, S. Irani and C. Swamy. Optimal power-down strategies. *Proc. 45th Annual IEEE Symposium on Foundations of Computer Science*, 530-539, 2004.

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

[8] Y. Azar, A.Z. Broder and A.R. Karlin. On-line load balancing. *Theoretical Computer Science*, 130(1):73–84, 1994.

[9] Y. Azar and L. Epstein. On-line load balancing of temporary tasks on identical machines. *SIAM Journal on Discrete Mathematics*, 18(2):347–352, 2004.

[10] Y. Azar, A. Epstein and L. Epstein. Load balancing of temporary tasks in the l_p norm. *Theoretical Computer Science*, 361(2-3):314–328, 2006.

[11] Y. Azar, B. Kalyanasundaram, S.A. Plotkin, K. Pruhs and O. Waarts. On-line load balancing of temporary tasks. *Journal of Algorithms*, 22(1):93–110, 1997.

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

0-8493-0052-5/00/$0.00+$.50

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

[14] N. Bansal, K. Pruhs and C. Stein. Speed scaling for weighted flow time. *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 805-813, 2007.

[15] L.A. Barroso. The price of performance. *ACM Queue*, 3(7), September 2005.

[16] Y. Bartal, A. Fiat, H. Karloff and R. Vohra. New algorithms for an ancient scheduling problem. *Journal of Computer and System Sciences*, 51:359–366, 1995.

[17] L. Becchetti and S. Leonardi. Nonclairvoyant scheduling to minimize the total flow time on single and parallel machines. *Journal of the ACM*, 51(4):517–539, 2004.

[18] D.P. Bunde. Power-aware scheduling for makespan and flow. *Proc. 18th Annual ACM Symposium on Parallel Algorithms and Architectures*, 190–196, 2006.

[19] 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. *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 795–804, 2007.

[20] C. Chekuri, S. Khanna and A. Zhu. Algorithms for minimizing weighted flow time. *Proc. 33rd Annual ACM Symposium on Theory of Computing*, 84–93, 2001.

[21] B. Chen, A. van Vliet and G.J. Woeginger. A lower bound for randomized on-line scheduling algorithms. *Information Processing Letters*, 51:219–222, 1994.

[22] U. Faigle, W. Kern and G. Turan. On the performance of on-line algorithms for particular problems. *Acta Cybernetica*, 9:107–119, 1989.

[23] R. Fleischer and M. Wahl. Online scheduling revisited. *Journal of Scheduling*, 3:343–353, 2000.

[24] G. Galambos and G. Woeginger. An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling. *SIAM Journal on Computing*, 22:349–355, 1993.

[25] B. Gaujal and N. Navet. Dynamic voltage scaling under EDF revisited. *Real-Time Systems*, 37(1):77–97, 2007.

[26] B. Gaujal, N. Navet and C. Walsh. Shortest path algorithms for real-time scheduling of FIFO tasks with optimal energy use. *ACM Transactions on Embedded Computing Systems*, 4(4):907-933, 2005.

[27] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[28] S. Irani, R.K. Gupta and S.K. Shukla. Competitive analysis of dynamic power management strategies for systems with multiple power savings states. *Proc. Design, Automation and Test in Europe, Conference and Exposition*, 117–123, 2002.

[29] S. Irani and A.R. Karlin. Online computation. In *Approximation Algorithms for NP-Hard Problems*, D. Hochbaum (ed.), 521–564, 1997.

[30] S. Irani and K. Pruhs. Algorithmic problems in power management. *ACM SIGACT News*, 36(2):63–76, 2005.

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

[32] B. Kalyanasundaram and K. Pruhs. Minimizing flow time nonclairvoyantly.. *Journal of the ACM*, 50(4):551–567, 2003.

[33] D.R. Karger, S.J. Phillips and E. Torng. A better algorithm for an ancient scheduling problem. *Journal of Algorithms*, 20:400–430, 1996.

[34] A.R. Karlin, M.S. Manasse, L.A. McGeoch and S.S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11(6):542–571, 1994.

[35] S. Leonardi. A simper proof of preemptive total flow time approximation on parallel machines. In *Approximation and Online Algorithms*, Springer LNCS, 2003.

[36] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. *Proc. 29th ACM Symposium on Theory of Computing*, 110–119, 1997.

[37] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. *Journal of Compututer and System Sciences*, 73(6):875–891, 2007.

[38] M. Li and F.F. Yao. An efficient algorithm for computing optimal discrete voltage schedules. *SIAM Journal on Computing*, 35(3):658–671, 2005.

[39] J. McCullough and E. Torng. SRPT optimally utilizes faster machines to minimize flow time. *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, 350–358, 2004.

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

[41] S. Muthukrishnan, R. Rajaraman, A. Shaheen and J. Gehrke. Online scheduling to minimize average stretch. *SIAM Journal on Computing*, 34(2):433–452, 2004.

[42] K. Pruhs, J. Sgall and E. Torng. Online Scheduling. In *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, J.Y-T. Leung (ed.), Chapter 15, 2007.

[43] K. Pruhs, P. Uthaisombut and G.J. Woeginger. Getting the best response for your erg. *Proc. 9th Scandinavian Workshop on Algorithm Theory (SWAT)*, Springer LNCS 3111, 14–25, 2004.

[44] J.F. Rudin III. Improved bounds for the on-line scheduling problem. Ph.D. Thesis. The University of Texas at Dallas, May 2001.

[45] S.S. Seiden. Online randomized multiprocessor scheduling. *Algorithmica*, 28:73–216, 2000.

[46] J. Sgall. A lower bound for randomized on-line multiprocessor scheduling. *Information Processing Letters*, 63:51–55, 1997.

[47] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules, *Communications of the ACM*, 28:202-208, 1985.

[48] F. Yao, A. Demers and S. Shenker. A scheduling model for reduced CPU energy. *Proc. 36th Annual Symposium on Foundations of Computer Science*, 374–382, 1995.