An Efficient Algorithm for Computing Optimal Discrete Voltage Schedules

Minming Li[1],⋆ and Frances F. Yao[2]

[1] State Key Laboratory of Intelligent Technology and Systems,
Dept. of Computer Science and Technology, Tsinghua Univ., Beijing, China
`liminming98@mails.tsinghua.edu.cn`

[2] Department of Computer Science, City University of Hong Kong
`csfyao@cityu.edu.hk`

Abstract. We consider the problem of job scheduling on a variable voltage processor with d discrete voltage/speed levels. We give an algorithm which constructs a minimum energy schedule for n jobs in $O(dn \log n)$ time. Previous approaches solve this problem by first computing the optimal continuous solution in $O(n^3)$ time and then adjusting the speed to discrete levels. In our approach, the optimal discrete solution is characterized and computed directly from the inputs. We also show that $O(n \log n)$ time is required, hence the algorithm is optimal for fixed d.

1 Introduction

Advances in processor, memory, and communication technologies have enabled the development and widespread use of portable electronic devices. As such devices are typically powered by batteries, energy efficiency has become an important issue. With dynamic voltage scaling techniques (DVS), processors are able to operate at a range of voltages and frequencies. Since energy consumption is at least a quadratic function of the supply voltage (hence CPU speed), it saves energy to execute jobs as slowly as possible while still satisfying all timing constraints.

We refer to the associated scheduling problem as min-energy DVS scheduling problem (or DVS problem for short); the precise formulation will be given in Section 2. The problem is different from classical scheduling on fixed-speed processors, and it has received much attention from both theoretical and engineering communities in recent years. One of the earliest theoretical models for DVS was introduced in [1]. They gave a characterization of the min-energy DVS schedule and an $O(n^3)$ algorithm [1] for computing it . No special assumption was made on the power consumption function except convexity. This optimal schedule has

⋆ This work is supported by National Natural Science Foundation of China (60135010), National Natural Science Foundation of China (60321002) and the Chinese National Key Foundation Research & Development Plan (2004CB318108).

[1] The complexity of the algorithm was said to be further reducible in [1], but that claim has since been withdrawn.

been referenced widely, since it provides a main benchmark for evaluating other scheduling algorithms in both theoretical and simulation work.

In the min-energy DVS schedule mentioned above, the processor must be able to run at *any* real-valued speed s in order to achieve optimality. In practice, variable voltage processors only run at a finite number of speed levels chosen from specific points on the power function curve. For example, the Intel *SpeedStep* technology [2] currently used in Intel's notebooks supports only 3 speed levels, although the new *Foxon* technology will soon enable Intel server chips to run at as many as 64 speed grades. Thus, an accurate model for min-energy scheduling should capture the discrete, rather than continuous, nature of the available speed scale. This consideration has motivated our present work.

In this paper we consider the discrete version of the DVS scheduling problem. Denote by $s_1 > s_2 > \ldots > s_d$ the clock speeds corresponding to d given discrete voltage levels. The goal is to find, under the restriction that only these speeds are available for job execution, a schedule that consumes as little energy as possible. (It is assumed that the highest speed s_1 is fast enough to guarantee a feasible schedule for the given jobs.) This problem was considered in [3] for a single job (i.e. $n = 1$), where they observed that minimum energy is achieved by using the immediate neighbors s_i, s_{i+1} of the ideal speed s in appropriate proportions. It was later extended in [4] to give an optimal discrete schedule for n jobs, obtained by first computing the optimal continuous DVS schedule, and then individually adjusting the speed of each job appropriately to adjacent levels as done in [3].

The question naturally arises: Is it possible to find a direct approach for solving the optimal discrete DVS scheduling problem, without first computing the optimal continuous schedule? We answer the question in the affirmative. For n jobs with arbitrary arrival-time/deadline constraints and d given discrete supply voltages (speeds), we give an algorithm that finds an optimal discrete DVS schedule in $O(dn \log n)$ time. We also show that this complexity is optimal for any fixed d. We remark that the $O(n^3)$ algorithm for finding continuous DVS schedule (cf Section 2) computes the highest speed, 2nd highest speed, etc for execution in a strictly sequential manner, and may use up to n different speeds in the final schedule. Therefore it is unclear a priori how to find shortcuts to solve the discrete problem. Our approach is different from that of [4] which is based on the continuous version and therefore requires $O(n^3)$ time.

Our algorithm for optimal discrete DVS proceeds in two stages. In stage 1, the jobs in J are partitioned into d disjoint groups $J_1, J_2, \ldots, J_d$ where J_i consists of all jobs whose execution speeds in the continuous optimal schedule S_{opt} lie between s_i and s_{i+1}. We show that this multi-level partition can be obtained without determining the exact optimal execution speed of each job. In stage two, we proceed to construct an optimal schedule for each group J_i using two speeds s_i and s_{i+1}. Both the separation of each group J_i in stage 1, and the subsequent scheduling of J_i using two speed levels in stage 2 can be accomplished in time $O(n \log n)$ per group. Hence this two-stage algorithm yields an optimal discrete voltage schedule for J in total time $O(dn \log n)$. The algorithm admits a simple implementation although its proof of correctness and complexity analysis are non-trivial. Aside from its theoretical value, we also expect our algorithm to be useful in generating optimal discrete DVS schedules for simulation purposes as in the continuous case.

We briefly mention some additional theoretical results on DVS, although they are not directly related to the problem considered in this paper. In [1], two on-line heuristics AVR (Average Rate) and OPA (Optimal Available) were introduced for the case that jobs arrive one at a time. AVR was shown to have a competitive ratio of at most 8 in [1]; recently a tight competitive ratio of 4 was proven for OPA in [5]. For jobs with fixed priority, the scheduling problem is shown to be NP-hard and an FPTAS is given in [6]. In addition, [7] gave efficient algorithms for computing the optimal schedule for job sets structured as trees. (Interested reader can find further references from these papers.)

The remainder of the paper is organized as follows. We give the problem formulation and review the optimal continuous schedule in Sections 2. Section 3 discusses some mathematical properties associated with EDF (earliest deadline first) scheduling under different speeds. Section 4 and Section 5 give details of the two stages of the algorithm as outlined above. The combined algorithm and a lower bound are presented in Section 6. Finally some concluding remarks are given in Section 7. Due to the page limit, many of the proofs are omitted in this version.

2 Problem Formulation

Each job j_k in a job set J over $[0, 1]$ is characterized by three parameters: arrival time a_k, deadline b_k and required number of CPU cycles R_k. A schedule S for J is a pair of functions $(s(t), job(t))$ defining the processor speed and the job being executed at time t. Both functions are piecewise constant with finitely many discontinuities. A *feasible* schedule must give each job its required number of cycles between arrival time and deadline (with perhaps intermittent execution). We assume that the power P, or energy consumed per unit time, is a convex function of the processor speed. The total energy consumed by a schedule S is $E(S) = \int_0^1 P(s(t))dt$. The goal of the min-energy scheduling problem is to find, for any given job set J, a feasible schedule that minimizes $E(S)$. We refer to this problem as *DVS* scheduling (or sometimes *Continuous DVS* scheduling to distinguish it from the discrete version below).

In the discrete version of the problem, we assume d discrete voltage levels are given, enabling the processer to run at d clock speeds $s_1 > s_2 > \ldots > s_d$. The goal is to find a minimum-energy schedule for a job set using only these speeds. We may assume that, in each problem instance, the highest speed s_1 is always fast enough to guarantee a feasible schedule for the given jobs. We refer to this problem as *Discrete DVS* scheduling.

For the continuous DVS scheduling problem, the optimal schedule S_{opt} can be characterized based on the notion of a *critical interval* for J, which is an interval I in which a group of jobs must be scheduled at maximum constant speed $g(I)$ in any optimal schedule for J. The algorithm proceeds by identifying such a critical interval I, scheduling those 'critical' jobs at speed $g(I)$ over I, then constructing a subproblem for the remaining jobs and solving it recursively. The optimal $s(t)$ is in fact unique, whereas $job(t)$ is not always so. The details are given below.

Definition 1. *Define the intensity of an interval* $I = [z, z']$ *to be* $g(I) = \frac{\sum R_j}{z'-z}$ *where the sum is taken over all jobs* j_ℓ *with* $[a_\ell, b_\ell] \subseteq [z, z']$*.*

The interval $[c, d]$ achieving the maximum $g(I)$ will be the critical interval chosen for the current job set. All jobs $j_\ell \in J$ satisfying $[a_\ell, b_\ell] \subseteq [c, d]$ can be feasibly scheduled at speed $g([c, d])$ by EDF principle. The interval $[c, d]$ is then removed from $[0, 1]$; all remaining intervals $[a_j, b_j]$ are updated (compressed) accordingly and the algorithm recurses. The complete algorithm is give in Algorithm 1.

```
Input: a job set J
Output: Optimal Voltage Schedule S
  repeat
    Select $I^* = [z, z']$ with $s = \max g(I)$
    Schedule $j_i \in J_{I^*}$ at $s$ over $I^*$ by Earliest Deadline First policy
    $J \leftarrow J - J_{I^*}$
    for all $j_k \in J$ do
      if $b_k \in [z, z']$ then
        $b_k \leftarrow z$
      else if $b_k \geq z'$ then
        $b_k \leftarrow b_k - (z' - z)$
      end if
      Reset arrival times similarly
    end for
  until J is empty
```

Algorithm 1: *OS (Optimal Schedule)*

Let $CI_i \subseteq [0, 1]$ be the ith critical interval of J. Denote by Cs_i the execution speed during CI_i, and by CJ_i those jobs executed in CI_i. We take note of a basic property of critical intervals which will be useful in later discussions.

Lemma 1. *A job* $j_\ell \in J$ *belongs to* $\bigcup_{k=1}^{i} CJ_k$ *if and only if the interval* $[a_\ell, b_\ell]$ *of* j_ℓ *satisfies* $[a_\ell, b_\ell] \subseteq \bigcup_{k=1}^{i} CI_k$*.*

3 EDF with Variable Speeds

The EDF (earliest deadline first) principle defines an ordering on the jobs according to their deadlines. At any time t, among jobs j_k that are available for execution, that is, j_k satisfying $t \in [a_k, b_k)$ and j_k is not yet finished by t, it is the job with minimum b_k that will be executed during $[t, t + \epsilon]$. The EDF is a natural scheduling principle and many optimal schedules (such as the continuous min-energy schedule described above) in fact conform to it. All schedules considered in the remainder of this paper are EDF schedules. Hence we assume the jobs in $J = \{j_1, \ldots, j_n\}$ are indexed by their deadlines.

We introduce an important tool for solving Discrete DVS scheduling problem: an EDF schedule that runs at some constant speed s (except for periods of idleness).

Definition 2. *An s-schedule for J is a schedule which conforms to EDF principle and uses constant speed s in executing any job of J.*

As long as there are unfinished jobs available at time t, an s-schedule will select a job by EDF principle and execute it at speed s. An s-schedule may contain periods of idleness when there are no jobs available for execution. An s-schedule may also yield an unfeasible schedule for J since the speed constraint may leave some jobs unfinished by deadline.

Definition 3. *In any schedule S, a maximal subinterval of $[0, 1]$ devoted to executing the same job j_k is called an execution interval (for j_k with respect to S). Denote by $I_k(S)$ the collection of all execution intervals for j_k with respect to S. With respect to the s-schedule for J, any execution interval will be called an s-execution interval, and the collection of all s-execution intervals for job j_k will be denoted by I_k^s.*

Notice that for any EDF schedule S, it is always true that $I_i(S) \subseteq [a_i, b_i] - \cup_{k=1}^{i-1} I_k(S)$. For a given J, we observe some interesting monotone relations that exist among the EDF schedules of J with respect to different speed functions. These relations will be exploited by our algorithms later. They may also be of independent interest in studying other types of scheduling problems.

Lemma 2. *Let S_1 and S_2 be two EDF schedules whose speed functions satisfy $s_1(t) > s_2(t)$ for all t whenever S_1 is not idle.*
1) For any t and any job j_k, the workload of j_k executed by time t under S_1 is always no less than that under S_2.
2) $\cup_{k=1}^{i} I_k(S_1) \subseteq \cup_{k=1}^{i} I_k(S_2)$ for any i, $1 \le i \le n$.
3) Any job of J that can be finished under S_2 is always finished strictly earlier under S_1.
4) If S_2 is a feasible schedule for J, then so is S_1.

Note that as a special case, Lemma 2 holds when we substitute s_1-schedule and s_2-schedule, with $s_1 > s_2$, for S_1 and S_2 respectively.

Lemma 3. *The s-schedule for J contains at most $2n$ s-execution intervals and can be computed in $O(n)$ time if the arrival times and deadlines are sorted.*

4 Partition of Jobs by Speed Level

We will consider the first stage of the algorithm in this section. Clearly, to obtain an $O(dn \log n)$-time partition of J into d groups corresponding to d speed levels, it suffices to give an $O(n \log n)$ algorithm which can properly separate J into two groups according to any given speed s.

Definition 4. *Given a job set J and any speed s, let $J^{\ge s}$ and $J^{<s}$ denote the subset of J consisting of jobs whose executing speed are $\ge s$ and $< s$ respectively in the (continuous) optimal schedule of J. We refer to the partition $\langle J^{\ge s}, J^{<s} \rangle$ as the s-*partition *of J.*

Let $T^{\geq s} \subseteq [0,1]$ be the union of all critical intervals CI_i with $Cs_i \geq s$. By Lemma 1, a job i is in $J^{\geq s}$ if and only if its interval $[a_i, b_i] \subseteq T^{\geq s}$. Thus $J^{\geq s}$ is uniquely determined by $T^{\geq s}$ and we can focus on computing $T^{\geq s}$ instead. Let $T^{<s} = [0,1] - T^{\geq s}$ and we refer to $\langle T^{\geq s}, T^{<s} \rangle$ as the *s-partition of time* for J.

An example of J with 11 jobs is given in Figure 1, together with the optimal speed function $S_{opt}(t)$. The portion of $S_{opt}(t)$ lying above the horizontal line $Y = s$ projects to $T^{\geq s}$ on the time-axis. In general, $T^{\geq s}$ may consist of a number of connected components.

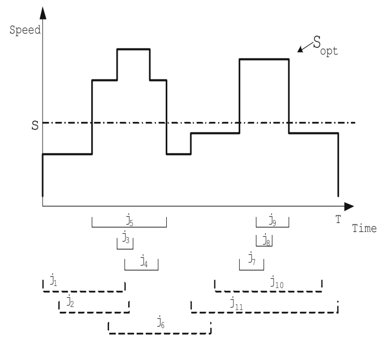
In the remainder of this section, we will show that certain features existing in the s-schedule of J can be used for identifying connected components of $T^{<s}$. This then leads to an efficient algorithm for computing the s-partition of time $\langle T^{\geq s}, T^{<s} \rangle$.

Definition 5. *In the s-schedule for J, we say a deadline b_i is tight if job j_i is either unfinished at time b_i, or it is finished just on time at b_i. An idle interval $g = [t, t']$ in the s-schedule is called a gap.*

Figure 2 depicts the s-schedule for the sample job set J considered in Figure 1. All tight deadlines and gaps have been marked along the time axis. By overlaying the s-partition of time $\langle T^{\geq s}, T^{<s} \rangle$ for J, we notice that 1) tight deadlines only exist in $T^{\geq s}$, and 2) each connected component of $T^{\geq s}$ ends with a tight deadline. We prove below that these properties always hold for any job set.

Lemma 4.

1) *Tight deadlines in an s-schedule can only exist in $T^{\geq s}$.*
2) *The rightmost point of each connected component of $T^{\geq s}$ must be a tight deadline.*



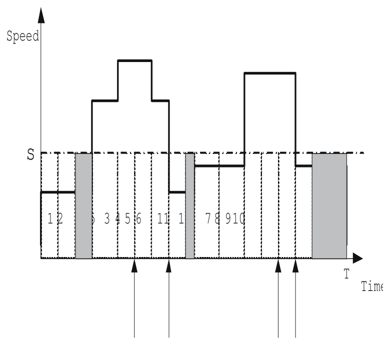


Fig. 1. The s-partition for a sample J. The jobs are represented by their intervals only, and indexed according to deadline. Solid intervals represent jobs belonging to $J^{\geq s}$, while dashed intervals represent jobs belonging to $J^{<s}$.

Fig. 2. The s-execution intervals for the same J in Fig. 1 are illustrated, where the number indicates which job is being executed. Shaded blocks correspond to gaps (idle time), while arrows point to tight deadlines.

Property 2) of Lemma 4 gives a necessary condition for identifying the right boundary of each connected component of $T^{\geq s}$. The corresponding left boundary of such a component can also be identified through left-right symmetry of the scheduling problem with respect to time.

Definition 6. *Given a job set J, the reverse job set J^{rev} consists of jobs with the same workload but time intervals $[1-b_i, 1-a_i]$. The s-schedule for J^{rev} is called the reverse s-schedule for J. We call an arrival time a_i (for the original job set J) tight if $1-a_i$ corresponds to a tight deadline in the reverse s-schedule for J.*

One may also view the reverse s-schedule as a schedule which runs backwards: starting from time 1 and executing jobs of J by the LAF principle (Latest Arrival time First) at constant speed s whenever possible.) Lemma 5 is the symmetric analogue of Lemma 4.

Lemma 5.

1) *Tight arrival times in an s-schedule can only exist in $T^{\geq s}$.*
2) *The leftmost point of each connected component of $T^{\geq s}$ must be a tight arrival time.*

Lemmas 4 and 5 are not sufficient by themselves to enable an efficient separation of $T^{\geq s}$ from $T^{<s}$. Fortunately, we have an additional useful property related to $T^{<s}$. Observe that in Figure 2 all gaps of the s-schedule fall within $T^{<s}$. This is in fact true in general and furthermore, a gap must exist in $T^{<s}$.

Lemma 6. *Gaps in an s-schedule can only exist in $T^{<s}$; furthermore a gap must exist in $T^{<s}$.*

Finally we collect the properties that will be used by the partition algorithm in the following theorem. We first give a definition.

Definition 7. *Given a gap $[x, y]$ in an s-schedule, we define the expansion of $[x, y]$ to be the smallest interval $[b, a]$ satisfying 1) $[b, a] \supseteq [x, y]$, and 2) b and a are tight deadline and tight arrival time respectively of the s-schedule. (Note: we adopt the convention that 0 is considered a tight deadline while 1 is considered a tight arrival time.)*

Theorem 1.

1) *A gap always exists in an s-schedule if $T^{<s} \neq \emptyset$.*
2) *The expansion $[b, a]$ of a gap $[x, y]$ defines the connected component in $T^{<s}$ containing $[x, y]$.*

Proof. Properties 1) comes from Lemma 6, while Property 2) follows from Lemma 4 and Lemma 5. □

Notice that, although Theorem 1 guarantees that one can always find a gap and then use it to identify a connected component C of $T^{<s}$ (provided $T^{<s} \neq \emptyset$), it is not true that *all* connected component of $T^{<s}$ must contain gaps and can

be identified simultaneously. However, once a component C is found, by deleting the s-execution intervals of all jobs whose interval $[a_i, b_i]$ intersects with C, gaps can surely be found (provided $T^{<s} - C \neq \emptyset$) and the process can continue. This is true because, by reasoning similar to that of Lemma 6, the total workload of the remaining jobs in $J^{<s}$ over $T^{<s} - C$ is less than $s \cdot |T^{<s} - C|$, hence a gap must exist.

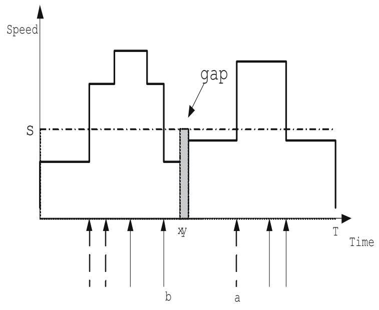


Fig. 3. Gap expansion: the indicated gap will be expanded into $[b, a]$, a connected component of $T^{<s}$

The detailed algorithm for generating the s-partition is given in Algorithm 2 below.

```
Input: job set $J$ and speed $s$
Output: $s$-partition of $J$
  Sort arrival times and deadlines
  Generate the $s$-schedule and reverse $s$-schedule for $J$
  $J^{\geq s} \leftarrow J$
  $J^{<s} \leftarrow \emptyset$
  $T^{\geq s} \leftarrow [0, 1]$
  $T^{<s} \leftarrow \emptyset$
  $Gaps$ = sorted list of gaps in $s$-schedule
  while $Gaps \neq \emptyset$ do
    1. Choose any gap $[x, y]$ from $Gaps$. Find the expansion $[b, a]$ of $[x, y]$.
    2.$J^{<s}_{new}$ = {all jobs in $J^{\geq s}$ whose interval $[a_j, b_j]$ intersects with $[b, a]$ }
    3.$J^{\geq s} \leftarrow J^{\geq s} - J^{<s}_{new}$
    4.$J^{<s} \leftarrow J^{<s} \cup J^{<s}_{new}$
    5.$T^{\geq s} \leftarrow T^{\geq s} - [b, a]$
    6.$T^{<s} \leftarrow T^{<s} \cup [b, a]$
    7.$Gaps = Gaps \cup \{$ $s$-execution intervals of jobs in $J^{<s}_{new} \}$
    8.Delete all gaps that are contained in $[b, a]$
  end while
  Return $J^{<s}$ and $J^{\geq s}$
```

Algorithm 2: Bi-Partition

Theorem 2. *Algorithm 2 finds the s-partition $\langle J^{\geq s}, J^{<s} \rangle$ for a job set J in $O(n \log n)$ time.*

Proof. The correctness of the algorithm is based on Theorem 1 and the discussions following the theorem. For the complexity part, sorting and generating s-schedules take $O(n \log n)$ time. We now analyze individual steps inside the for loop. For Step 1, finding the expansion of a gap only takes $O(\log n)$ time by binary search; with at most n expansions (to find at most n connected components) the total cost is $O(n \log n)$. Step 2 can be done, with standard data structures such as interval trees, in time $O(\log n) + |J_{new}^{<s}|$ which amounts to total time $O(n \log n)$ since $\sum |J_{new}^{<s}| = O(n)$. It remains to consider the cost of Steps 7 and 8. Since each individual gap is added to and deleted from the sorted list *Gaps* only once, and there are at most $2n$ s-execution intervals (hence gaps), the cost is at most $O(n \log n)$. This shows that the total running time of Algorithm 2 is $O(n \log n)$. □

We next use Algorithm 2 as a subroutine to obtain Algorithm 3.

```
Input:
job set $J$ and speed $s_1 > \ldots > s_d > s_{d+1} = 0$
Output:
Partition of $J$ into $J_1, \ldots, J_d$ corresponding to speed levels
  for $i = 1$ to $d$ do
    Obtain $J^{\geq s_{i+1}}$ from $J$ using Algorithm 2
    $J_i \leftarrow J^{\geq s_{i+1}}$
    $J \leftarrow J - J_i$
    Update $J$ as in Algorithm 1
  end for
```

Algorithm 3: Multi-level Partition

Theorem 3. *Algorithm 3 partitions job set J into d subsets corresponding to d speed levels in time $O(dn \log n)$.*

5 Two-Level Schedule

After Algorithm 3 completes the multi-level partition of J into subsets $J_1, \ldots, J_d$, we can proceed to schedule the jobs in each subset J_i with two appropriate speed levels s_i and s_{i+1}. We will present a two-level scheduling algorithm whose complexity is $(n \log n)$ for a set of n jobs. For this purpose, it suffices to describe how to schedule the subset J_1 with two available speeds s_1 and s_2 where $s_1 > s_2 > 0$. We will schedule each connected component of J_1 separately. Thus, the two-level scheduler only deals with 'eligible' input job sets, i.e., those with a continuous optimal schedule speed $s_{opt}(t)$ satisfying $s_1 \geq s_{opt}(t) \geq s_2$ for all t. (Clearly, this condition is satisfied by each connected component of $J_1 = J^{\geq s_2}$ output from Algorithm 3.) We give an alternative and equivalent definition of 'eligibility' in the following. This definition does not make reference to $s_{opt}(t)$ and hence is more useful for the purpose of deriving a two-level schedule directly.

Definition 8. *For a job set J over $[0, 1]$, a two-level schedule with speeds s_1 and s_2 (or (s_1, s_2)-schedule for short) for J is a feasible schedule $s(t)$ for J, which is piecewise constant over $[0, 1]$ with either $s(t) = s_1$ or $s(t) = s_2$ for any t.*

In other words, an (s_1, s_2)-schedule for J is a schedule using only speeds s_1 and s_2 which finishes every job and leaves no idle time.

Lemma 7. *For a job set J over $[0, 1]$, an (s_1, s_2)-schedule exists if and only if*
1) the s_1-schedule for J is a feasible schedule, and
2) the s_2-schedule for J contains no idle time in $[0, 1]$.

In view of the preceding lemma, we give the following definition of eligibility for input job sets to two-level scheduling.

Definition 9. *A job set J over $[0, 1]$ is said to be eligible for (s_1, s_2)-scheduling if 1) the s_1-schedule for J is a feasible schedule, and*
2) the s_2-schedule for J contains no idle time in $[0, 1]$.

We will consider only eligible job sets in discussing two-level scheduling in the remainder of this section. An (s_1, s_2)-schedule for J is said to be *optimal* if it consumes minimum energy among all (s_1, s_2)-schedules for J.

Lemma 8. *All (s_1, s_2)-schedules for an eligible job set J consume the same amount of energy and hence are optimal.*

The two-level schedule as described in the proof of Lemma 7, which first computes the continuous optimal schedule and then rounds the execution speed of each job up and down appropriately [4], requires $O(n^3)$ computation time. We now describe a more efficient algorithm which directly outputs a two-level schedule without first computing the continuous optimal schedule. The algorithm runs in $O(n)$ time if the input jobs are already sorted by deadline (as obtained via Multi-level Partition), and $O(n \log n)$ time in general.

The two-level scheduling algorithm (Algorithm 4) proceeds as follows. It first computes the s_2-schedule for J which in general does not provide a feasible schedule. We then transform it into a feasible schedule by suitably adjusting the execution speed of each job from s_2 to s_1, and possibly extending its execution interval if necessary. These adjustments are done in an orderly and systematic manner to ensure overall feasibility. The algorithm needs to consult the corresponding s_1-schedule of J in making the transformation. An (s_1, s_2)-schedule for J is produced at the end which by lemma 8 is an optimal two-level schedule.

```
Input:
speeds $s_1, s_2$ where $s_1 > s_2$
An eligible job set $J$ for $(s_1, s_2)$-scheduling
Variables:
Committed: the list of allocated time intervals.
Committed(i): the time intervals allocated to job $j_i$.

Output:
Optimal $(s_1, s_2)$-schedule for $J$
  Compute $s_1$-schedule for $J$ to obtain $I_k^{s1}$ for $k = 1, \ldots, n$.
  Compute $s_2$-schedule for $J$ to obtain $I_k^{s2}$ for $k = 1, \ldots, n$.
  $Committed \leftarrow \emptyset$
  for $i = n$ downto 1 do
    1. I $= I_i^{s2} - Committed$
    2. Take $I' \subseteq I_i^{s1}$ of appropriate length (possibly 0) from the right end of $I_i^{s1}$
       to obtain an $(s_1, s_2)$-schedule for $j_i$ over $I \cup I'$
    3. $Committed(i) = I \cup I'$
    4. $Committed \leftarrow Committed \cup Committed(i)$
  end for
```

Algorithm 4: Two-Level Schedule

In the remainder of this section, we consider the correctness and complexity of Algorithm 4.

Let J be an eligible job set for (s_1, s_2)-scheduling. Assume the jobs in J are sorted in increasing order by their deadlines as $j_1, j_2, \ldots, j_n$. After computing the s_1-schedule and s_2-schedule for J, the algorithm then allocates appropriate execution time and speed for each job j_i, in the order $i = n, \ldots, 1$. Step 2 of the for loop carries out the allocation for job j_i. We examine this step in more detail in the following lemma.

Lemma 9. *In step 2 of the for loop, by choosing an appropriate interval $I' \subseteq I_i^{s_1}$ (assuming $I_i^{s_1} \cap Committed = \emptyset$), an (s_1, s_2)-schedule for job j_i over $I \cup I'$ can be found where $I = I_i^{s_2} - Committed$.*

We next show that the assumption $I_i^{s_1} \cap Committed = \emptyset$ in Lemma 9 is indeed satisfied when step 2 is encountered in the i-th iteration (see property 3) below). In fact, we show by induction on i that the following induction hypotheses are maintained by the algorithm at the start of the i-th iteration for $i = n, \ldots, 1$.

Lemma 10. *At the beginning of the i-th iteration of the for loop, the following are true:*
1) $Committed(i+1) \subseteq I_{i+1}^{s_1} \cup I_{i+1}^{s_2}$
2) $\cup_{k=i+1}^{n} I_k^{s_2} \subseteq Committed \subseteq (\cup_{k=i+1}^{n} I_k^{s_1}) \cup (\cup_{k=i+1}^{n} I_k^{s_2})$
3) $Committed \cap (\cup_{k=1}^{i} I_k^{s_1}) = \emptyset$.

Theorem 4. *Given an eligible job set J for (s_1, s_2)-scheduling, Algorithm 4 generates an (s_1, s_2)-schedule for J.*

Proof. Each job j_i is feasibly executed, with no idle time, over $Committed(i)$ at speeds $\{s_1, s_2\}$ as specified in Lemma 9. By the time the algorithm terminates, $Committed = \cup_{k=1}^{n} Committed(k) \supseteq \cup_{k=1}^{n} I_k^{s_2} = [0, 1]$ by Property 2) of Lemma 10. Hence there is no idle time in $[0, 1]$. The resulting schedule thus satisfies the requirements of an (s_1, s_2)-schedule for J. □

Theorem 5. *Algorithm 4 computes an optimal two-level schedule for J in $O(n)$ time if the jobs in J are sorted by deadline (as output by Algorithm 3), and in $O(n \log n)$ time otherwise.*

6 Optimal Discrete Voltage Schedule

Theorem 6. *Algorithm 5 generates a min-energy Discrete DVS schedule with d voltage levels in time $O(dn \log n)$ for n jobs.*

Proof. This is a direct consequence of Theorem 3, Theorem 4 and Theorem 5. □

We next show that the running time of Algorithm 5 is optimal by proving an $\Omega(n \log n)$ lower bound in the algebraic decision tree model.

Theorem 7. *Any deterministic algorithm for computing a min-energy Discrete DVS schedule (MDDVS) with $d \geq 2$ voltage levels will require $\Omega(n \log n)$ time for n jobs.*

```
Input:
job set $J$
speed levels: $s_1 > s_2 > \ldots > s_d > s_{d+1} = 0$
Output:
Optimal Discrete DVS Schedule for $J$
  Generate $J_1, J_2, \ldots, J_d$ by Algorithm 3
  for $i = 1$ to $d$ do
    Schedule jobs in $J_i$ using Algorithm 4 with speeds $s_i$ and $s_{i+1}$
  end for
  The union of the schedules give the optimal Discrete DVS schedule for $J$
```

Algorithm 5: Optimal Discrete DVS Schedule

7 Conclusion

In this paper we considered the problem of job scheduling on a variable voltage processor with d discrete voltage/speed levels. We give an algorithm which constructs a minimum energy schedule for n jobs in $O(dn \log n)$ time, which is optimal for fixed d. The min-energy discrete schedule is obtained without first computing the continuous optimal solution. Our algorithm consists of two stages: a multi-level partition of J into d disjoint groups J_i, followed by finding a two-level schedule for each J_i using speeds s_i and s_{i+1}. The individual modules in our algorithm, such as the multi-level partition and two-level scheduling, may be of interest in themselves aside from the main result. Our algorithm admits a simple implementation although its proof of correctness and complexity analysis are non-trivial. We have also discovered some nice fundamental properties associated with EDF scheduling under variable speeds. Some of these properties are stated as lemmas in Section 3 for easy reference. Our results may provide some new insights and tools for the problem of min-energy job scheduling on variable voltage processors. Aside from the theoretical value, we also expect the algorithm to be useful in generating optimal discrete schedules for simulation purposes as in the continuous case.

References

1. F. Yao, A. Demers and S. Shenker, *A Scheduling Model for Reduced CPU Energy*, IEEE Proc. FOCS 1995, 374-382.
2. Intel Corporation, *Wireless Intel SpeedStep Power Manager - Optimizing Power Consumption for the Intel PXA27x Processor Family*, Wireless Intel SpeedStep(R) Power Manager White Paper, 2004.
3. T. Ishihara and H. Yasuura, *Voltage Scheduling Problem for Dynamically Variable Voltage Processors*, ISLPED, 1998.
4. W. Kwon and T. Kim, *Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors*, 40^{th} Design Automation Conference, 2003.
5. N. Bansal, T. Kimbrel and K. Pruhs, *Dynamic Speed Scaling to Manage Energy and Temperature*, IEEE Proc. FOCS 2004, 520-529.
6. H. S. Yun and J. Kim, *On Energy-Optimal Voltage Scheduling for Fixed-Priority Hard Real-Time Systems*, ACM Trans. Embedded Comput. Syst. 2(3):393-430, 2003.
7. M. Li, J. B. Liu and F. F. Yao, *Min-Energy Voltage Allocation for Tree-Structured Tasks*, to appear in COCOON 2005.
8. A. C. Yao, *Lower Bounds for Algebraic Computation Trees with Integer Inputs*, SIAM J. Comput. 20(1991):308-313.