

AN EFFICIENT ALGORITHM FOR COMPUTING OPTIMAL DISCRETE VOLTAGE SCHEDULES*

MINMING LI[†] AND FRANCES F. YAO[‡]

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 .

Key words. scheduling, energy efficiency, variable voltage processor, discrete optimization

AMS subject classifications. 90B35, 90B80

DOI. 10.1137/050629434

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 (DVS) techniques, 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 the 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 $O(n^3)$ algorithm¹ for computing it. No special assumption was made on the power consumption function except convexity. This optimal schedule has 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 run only 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

*Received by the editors April 18, 2005; accepted for publication (in revised form) August 15, 2005; published electronically January 6, 2006. This work was supported in part by the Research Grants Council of Hong Kong under grant CityU122105, the National Natural Science Foundation of China under grants 60135010 and 60321002, and the Chinese National Key Foundation Research & Development Plan (2004CB318108).

<http://www.siam.org/journals/sicomp/35-3/62943.html>

[†]Department of Computer Science and Technology, State Key Laboratory of Intelligent Technology and Systems, Tsinghua University, Beijing, China (liminming98@mails.tsinghua.edu.cn).

[‡]Department of Computer Science, City University of Hong Kong, Hong Kong, China (csfyao@cityu.edu.hk).

¹The complexity of the algorithm was said to be further reducible in [1], but that claim has since been withdrawn.

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 $s_1 > s_2 > \dots > s_d$ 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 following 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 the continuous DVS schedule (cf. section 2) computes the highest speed, second 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, \dots, 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 multilevel partition can be obtained without determining the exact optimal execution speed of each job. In stage 2, 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 nontrivial. 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, average rate (AVR) and optimal available (OPA), 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 was shown to be NP-hard, and an FPTAS was given in [6]. In addition, [7] gave efficient algorithms for computing the optimal schedule for job sets structured as trees. (The interested reader can find further references in these papers.)

The remainder of the paper is organized as follows. We give the problem formu-

lation and review the optimal continuous schedule in section 2. Section 3 discusses some mathematical properties associated with earliest deadline first (EDF) scheduling under different speeds. Sections 4 and 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.

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 processor to run at d clock speeds $s_1 > s_2 > \dots > s_d$. The goal is to find a min-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 and 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 2.1. 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 the 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 given in Algorithm 1.

Let $CI_i \subseteq [0, 1]$ be the i th 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 2.2. 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 $[a_\ell, b_\ell] \subseteq \bigcup_{k=1}^i CI_k$.

Proof. The “if” direction is straightforward. For the “only if” part, it can be proven by induction on i that $j_\ell \in CJ_i$ implies $[a_\ell, b_\ell] \subseteq \bigcup_{k=1}^i CI_k$, based on the way critical intervals are successively chosen. \square

Algorithm 1 *OS (Optimal Schedule)*

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

3. EDF with variable speeds. The EDF 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 not yet finished by t , it is the job with minimum b_k that will be executed during $[t, t + \epsilon]$. 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 $J = \{j_1, \dots, j_n\}$ are indexed by their deadlines.

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

DEFINITION 3.1. *An s -schedule for J is a schedule which conforms to the 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 the 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 their deadlines.

DEFINITION 3.2. *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.

Comparison between two s -schedules with different speeds.

LEMMA 3.3. *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 $i, 1 \leq i \leq n$.

(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 .

Proof. We prove (1) and (2) by induction on i . When $i = 1$, since both S_1 and S_2 start executing j_1 at time a_1 , it is easy to see that induction hypotheses (1) and (2) are both true. Assume that they hold for jobs j_1, \dots, j_{i-1} ; we will prove (1) and (2) for j_i . The time V_1 available for executing j_i under S_1 is $V_1 = [a_i, b_i] - \cup_{k=1}^{i-1} I_k(S_1)$, which satisfies $V_1 \supseteq V_2$ for the corresponding available time under S_2 because of induction hypothesis (2). This together with the assumption $s_1(t) > s_2(t)$ proves (1); that is, the execution of j_i by S_1 will always be ahead of that by S_2 . Assuming that S_2 finishes j_i at time t , then we have $I_i(S_1) \subseteq [a_i, t] \subseteq \cup_{k=1}^i I_k(S_2)$ from which (2) follows inductively. \square

Note that as a special case, Lemma 3.3 holds when we substitute s_1 -schedule and s_2 -schedule, with $s_1 > s_2$, for S_1 and S_2 , respectively.

LEMMA 3.4. *The s -schedule for J contains at most $2n$ s -execution intervals and can be computed in $O(n \log n)$ time if the arrival times and deadlines are already sorted.*

Proof. The end of an execution interval corresponds to the moment when either a job is finished or a new job arrives. There can be at most $2n$ such endpoints, and hence at most $2n$ s -execution intervals. If the arrival times and deadlines are already sorted, then generating one s -execution interval costs $O(\log n)$ time, and the entire schedule can be computed in $O(n \log n)$ time. This completes the proof. \square

4. Partition of jobs by speed level. We will consider the first stage of the algorithm for optimal discrete DVS in this section. Clearly, to get 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 4.1. *Given a job set J and any speed s , let $J^{\geq s}$ and $J^{< s}$ denote the subset of J consisting of jobs whose executing speeds are $\geq s$ and $< s$, respectively, in the (continuous) optimal schedule of J . We refer to the partition $\langle J^{\geq s}, J^{< s} \rangle$ 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 2.2, 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 4.2. *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 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

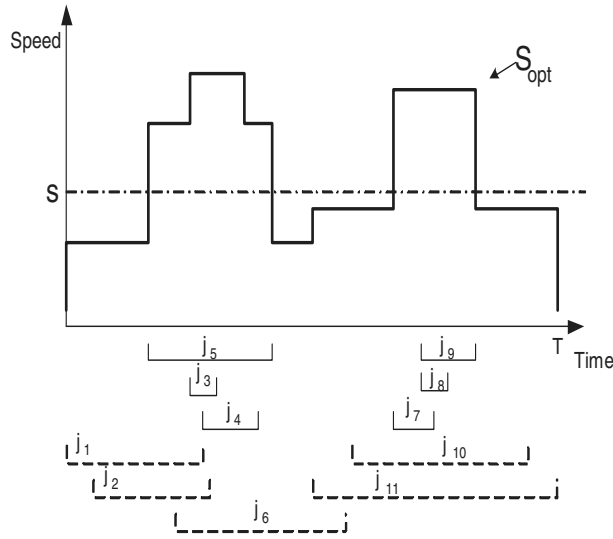


FIG. 1. The s -partition for a sample J . The jobs are represented by their intervals only and indexed according to deadline. Solid intervals represent jobs belonging to $J^{\geq s}$, while dashed intervals represent jobs belonging to $J^{< s}$.

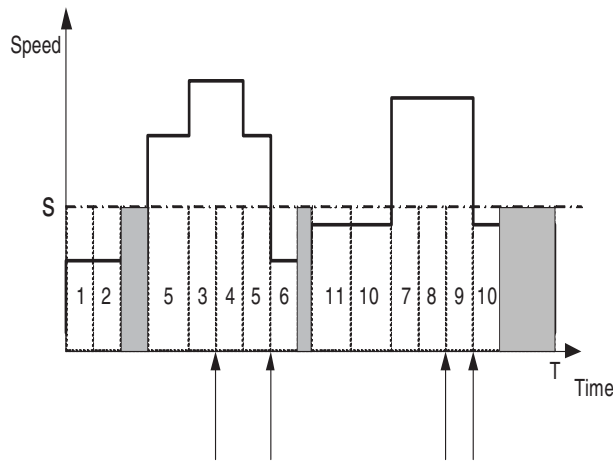


FIG. 2. The s -execution intervals for the same J in Figure 1 are illustrated, where the number indicates which job is being executed. Shaded blocks correspond to gaps (idle time), while arrows point to tight deadlines.

s -partition of time $\langle T^{\geq s}, T^{< s} \rangle$ for J , we notice that (1) tight deadlines exist only 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 4.3. (1) *Tight deadlines in an s -schedule can exist only in $T^{\geq s}$.*

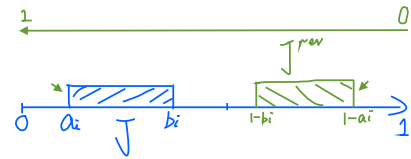
(2) *The rightmost point of each connected component of $T^{\geq s}$ must be a tight deadline.*

Proof. As observed before, the only jobs available for execution during $T^{< s}$ are those from $J^{< s}$. Furthermore, by property (3) of Lemma 3.3, all jobs of $J^{< s}$ will be finished strictly before their deadlines under the s -schedule, thus yielding no tight

deadlines in $T^{<s}$. This proves property (1). For property (2), we argue that it is a consequence of property (1) in Lemma 3.3. Let b_i be the end of a connected component of $T^{\geq s}$. Job j_i , being the last executed job of a critical interval, is finished exactly on time under S_{opt} (which runs at speed at least s throughout $T^{\geq s}$). Therefore, job j_i must have a tight deadline under the s -schedule. \square

Property (2) of Lemma 4.3 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 4.4. 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 latest arrival time first principle at constant speed s whenever possible. Lemma 4.5 is the symmetric analogue of Lemma 4.3.

LEMMA 4.5. (1) Tight arrival times in an s -schedule can exist only in $T^{\geq s}$.

(2) The leftmost point of each connected component of $T^{\geq s}$ must be a tight arrival time.

Lemmas 4.3 and 4.5 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 $T^{<s}$ as we prove next.

LEMMA 4.6. Gaps in an s -schedule can exist only in $T^{<s}$; furthermore, a gap must exist in $T^{<s}$.

By contradiction

Proof. Suppose a gap in an s -schedule occurs at some time $t \in T^{\geq s}$; that is, all jobs $J(t)$ in J whose intervals overlap t have been finished. In particular, no jobs belonging to $J(t) \cap J^{\geq s}$ are available. Since the schedule s_{opt} runs at higher speed than s over $T^{\geq s}$ in executing $J^{\geq s}$, it must also finish all jobs of $J(t) \cap J^{\geq s}$ before time t by Lemma 3.3. In other words, s_{opt} would have a gap at time t which is not possible. This proves that gaps can exist only in $T^{<s}$. For the second part, we note that the total workload of $J^{<s}$, which is executed over $T^{<s}$, is less than $s \cdot |T^{<s}|$; hence a gap must exist. \square

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

DEFINITION 4.7. 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.) See Figure 3.

THEOREM 4.8. (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. Property (1) comes from Lemma 4.6, while property (2) follows from Lemmas 4.3 and 4.5. \square

Notice that although Theorem 4.8 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 components of $T^{<s}$ must contain gaps and can be identified

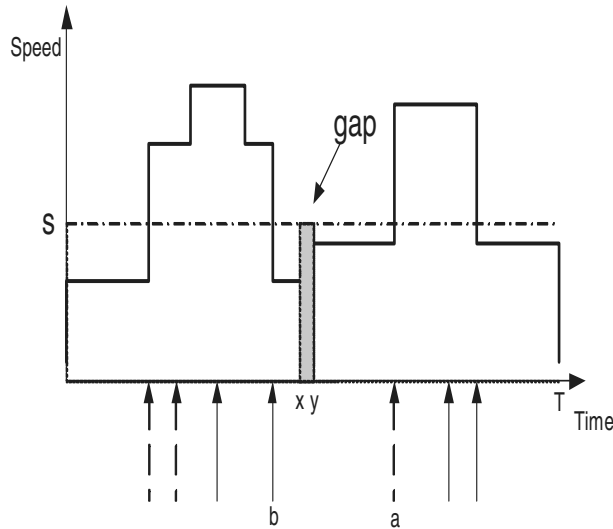


FIG. 3. Gap expansion: the indicated gap will be expanded into $[b, a]$, a connected component of $T^{<s}$.

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 4.6, 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.

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

THEOREM 4.9. *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 4.8 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 takes only $O(\log n)$ 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 steps 7 and 8. Since each individual gap is added to and deleted from the sorted list *Gaps* only once, and since 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 $O(n \log n)$. \square

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


THEOREM 4.10. *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 multilevel partition of J into subsets J_1, \dots, 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 $O(n \log n)$ 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

Algorithm 2 Bipartition**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. $J_{new}^{< s} = \{ \text{all jobs in } J^{\geq s} \text{ whose interval } [a_j, b_j] \text{ intersects with } [b, a] \}$
3. $J^{\geq s} \leftarrow J^{\geq s} - J_{new}^{< s}$ 
4. $J^{< s} \leftarrow J^{< s} \cup J_{new}^{< s}$
5. $T^{\geq s} \leftarrow T^{\geq s} - [b, a]$
6. $T^{< s} \leftarrow T^{< s} \cup [b, a]$
7. $Gaps = Gaps \cup \{ s\text{-execution intervals of jobs in } J_{new}^{< s} \}$
8. **Delete all gaps** that are contained in $[b, a]$

end whileReturn $J^{< s}$ and $J^{\geq s}$ **Algorithm 3** Multilevel Partition**Input:**job set J and speed $s_1 > \dots > s_d > s_{d+1} = 0$ **Output:**Partition of J into J_1, \dots, 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**

s_2 , where $s_1 > s_2 > 0$. We will schedule each connected component of J_1 separately. Thus, the two-level scheduler deals only 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 5.1. 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 5.2. 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 $[0, 1]$.

Proof. The “only if” direction is easy to see. Suppose a two-level schedule $s(t)$ exists with $s_1 \geq s(t) \geq s_2$ for all $t \in [0, 1]$. It follows from Lemma 3.3 that with speed s_1 the processor can finish all jobs just as with speed $s(t)$, while with speed s_2 the processor will not finish any job earlier than with speed $s(t)$, and hence will never be idle. This proves (1) and (2). For the “if” direction, suppose (1) and (2) both hold. Because the s_1 -schedule generates a feasible schedule, the optimal continuous schedule $s_{opt}(t)$ must satisfy $s_{opt}(t) \leq s_1$ for all $t \in [0, 1]$. On the other hand, (2) implies $J = J^{\geq s_2}$ by Lemma 4.6; that is, $s_{opt}(t) \geq s_2$ for all $t \in [0, 1]$. Using the result in [4], we can first calculate the continuous optimal schedule $s_{opt}(t)$ for J and then adjust the execution speed s of each job to be a combination of s_1 or s_2 in the right proportion to achieve the same average speed s . This results in a two-level schedule for J with speeds s_1 and s_2 . \square

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

DEFINITION 5.3. 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 5.4. All (s_1, s_2) -schedules for an eligible job set J consume the same amount of energy, and hence are optimal.

Proof. The energy consumption is determined by the total amount of time the processor runs at speeds s_1 and s_2 , respectively. Suppose, in an optimal schedule for J , that α time is devoted to speed s_1 and β time is devoted to speed s_2 . An optimal schedule will not contain any idle period; hence the following equations are satisfied:

$$\begin{cases} \alpha s_1 + \beta s_2 = \sum R_i, \\ \alpha + \beta = 1. \end{cases}$$

Clearly, any (s_1, s_2) -schedule for J will also satisfy the above two equations. Since these equations uniquely determine α and β , the lemma follows. \square

The two-level schedule as described in the proof of Lemma 5.2, 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 Algorithm 3) 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 5.4 is an optimal two-level schedule.

Algorithm 4 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 Subset of J shown above

Variables:

Committed: the list of allocated time intervals. Committed(i)

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 $I_k^{s_1}$ for $k = 1, \dots, n$.

Compute s_2 -schedule for J to obtain $I_k^{s_2}$ for $k = 1, \dots, n$.

$Committed \leftarrow \emptyset$

for $i = n$ **downto** 1 **do**

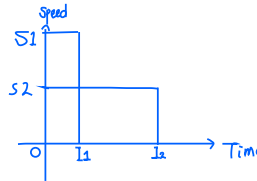
1. $I = I_i^{s_2} - Committed$

2. Take $I' \subseteq I_i^{s_1}$ of appropriate length (possibly 0) from the right end of $I_i^{s_1}$ to obtain an (s_1, s_2) -schedule for j_i over $I \cup I'$

3. $Committed(i) = I \cup I'$ extending its execution interval

4. $Committed \leftarrow Committed \cup Committed(i)$

end for



5.1. Correctness of two-level scheduling algorithm. Let J be an eligible job set for (s_1, s_2) -scheduling. We will show that Algorithm 4 indeed outputs an (s_1, s_2) -schedule for J .

The jobs in J are sorted in increasing order by their deadlines as j_1, j_2, \dots, 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 $i = n, \dots, 1$. 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 5.5. *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 $I = I_i^{s_2} - Committed$.*

Proof. There are two cases to consider when step 2 is encountered. Suppose job j_i can be feasibly scheduled with speed s_1 over $I = I_i^{s_2} - Committed$. Since the s_2 -schedule of j_i over I clearly has no idle time, Lemma 5.2 ensures that an (s_1, s_2) -schedule exists for job j_i over I . Suppose j_i cannot be feasibly scheduled at s_1 over I . Under the assumption $I_i^{s_1} \cap Committed = \emptyset$, we can take sufficient length of time I' from $I_i^{s_1}$ so that j_i can be finished at s_1 over $I \cup I'$. Therefore one can always find an (s_1, s_2) -schedule for job j_i over $I \cup I'$. □

We next prove that the assumption $I_i^{s_1} \cap Committed = \emptyset$ in Lemma 5.5 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 $i = n, \dots, 1$.

LEMMA 5.6. *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) $Committed \cap (\cup_{k=1}^i I_k^{s_1}) = \emptyset$.

Proof. It is easy to verify that all three induction hypotheses hold initially for $i = n$. Now assume that they hold for iterations $i+1, \dots, n$; we will prove them for the i th iteration. Property (1) is a result of how $Committed(i+1)$ is selected as discussed in Lemma 5.5. For property (2), the right side follows from (1) since $Committed = \cup_{k=i+1}^n Committed(k)$. The left side follows from the fact that, by Lemma 5.5, each $Committed(k)$ always uses up all remaining time in $I_k^{s_2}$ not already committed to previous jobs. To prove (3), let $V = \cup_{k=1}^i I_k^{s_1}$. First, note that V is disjoint from $\cup_{k=i+1}^n I_k^{s_1}$. Next, V is contained in $\cup_{k=1}^i I_k^{s_2}$ by property (2) of Lemma 3.3; hence V is disjoint from $\cup_{k=i+1}^n I_k^{s_2}$. Thus it follows from (2) that $Committed \cap V = \emptyset$. \square

THEOREM 5.7. *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 5.5. 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 5.6. 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 . \square

5.2. Complexity of two-level scheduling algorithm. We will show that the cost of Algorithm 4 is $O(n \log n)$. The algorithm first computes the s_1 -schedule and s_2 -schedule for J in $O(n \log n)$ time. The resulting list L_{s_1} with at most $2n$ s_1 -execution intervals is already sorted, and similarly for the list L_{s_2} of s_2 -execution intervals. A sorted list L_{s_1, s_2} , of size at most $4n$, representing $L_{s_1} \cap L_{s_2}$ can be obtained with cost $O(n)$. Using appropriate pointers from lists L_{s_1} , L_{s_2} , and $Committed$ into L_{s_1, s_2} , each step of the for loop can be carried out in constant time plus the number of intervals visited. Execution of step 2 may cause a splitting of some interval in L_{s_1} (to represent interval I') and corresponding splitting in L_{s_1} , L_{s_1, s_2} , and $Committed$. As only a single split can be introduced in each iteration, the overall effect is only $O(n)$. Also each subinterval in L_{s_1} , L_{s_2} , and L_{s_1, s_2} needs to be visited only once. Hence the total running time of the algorithm is $O(n)$ if the input jobs are sorted (as output by Algorithm 3). We have proved the following theorem.

THEOREM 5.8. *Algorithm 4 computes an optimal two-level schedule for J in $O(n \log n)$ time.*

Algorithm 5 Optimal Discrete DVS Schedule

Input:

job set J

speed levels: $s_1 > s_2 > \dots > s_d > s_{d+1} = 0$

Output:

Optimal Discrete DVS Schedule for J

Generate J_1, J_2, \dots, 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

6. Optimal discrete voltage schedule.

THEOREM 6.1. *Algorithm 5 generates a min-energy discrete DVS (MDDVS) schedule with d voltage levels in time $O(dn \log n)$ for n jobs.*

Proof. This is a direct consequence of Theorems 4.10, 5.7, and 5.8. \square

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 6.2. *Any deterministic algorithm for computing an MDDVS schedule with $d \geq 2$ voltage levels will require $\Omega(n \log n)$ time for n jobs.*

Proof. The integer element uniqueness (IEU) problem is known to have $\Omega(n \log n)$ computational complexity in the algebraic decision tree model [8]. We now make a **linear reduction** from IEU to MDDVS. Suppose the given instance of IEU consists of n positive integers $\{x_1, x_2, \dots, x_n\}$. First, compute $N = \max\{x_i\}$ in linear time. We construct a job set $J = \{j_1, j_2, \dots, j_n\}$ over time span $[0, N]$ with $[a_i, b_i] = [x_i - 1, x_i]$ and $R_i = 1$. (The time span can be normalized to $[0, 1]$ by scaling all numbers appropriately.) Thus the time intervals of all the jobs are disjoint if and only if the integers x_i are distinct. Set the available speed levels to be $s_1 = n$ (to guarantee feasibility) and $s_d = 1$, while s_2, \dots, s_{d-1} may be any values in between. It is easy to see that the answer to the IEU problem is yes (all integers x_i are distinct) if and only if $MDDVS \leq n$ (by executing every job at speed $s_d = 1$). This completes the reduction. \square

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 multilevel 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 multilevel partition and two-level scheduling, may be of interest in and of themselves aside from the main result. Our algorithm admits a simple implementation, although its proof of correctness and complexity analysis are nontrivial. 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*, in Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science, 1995, pp. 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*, in Proceedings of the International Symposium on Low Power Electronics and Design, ACM, New York, 1998, pp. 197–201.
- [4] W. KWON AND T. KIM, *Optimal voltage allocation techniques for dynamically variable voltage processors*, ACM Transactions on Embedded Computing Systems, 4 (2005), pp. 211–230.

- [5] N. BANSAL, T. KIMBREL, AND K. PRUHS, *Dynamic speed scaling to manage energy and temperature*, in Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science, 2004, pp. 520–529.
- [6] H. S. YUN AND J. KIM, *On energy-optimal voltage scheduling for fixed-priority hard real-time systems*, ACM Transactions on Embedded Computing Systems, 2 (2003), pp. 393–430.
- [7] M. LI, J. B. LIU, AND F. F. YAO, *Min-energy voltage allocation for tree-structured tasks*, in Proceedings of the Eleventh International Computing and Combinatorics Conference, Springer-Verlag, Berlin, 2005, pp. 283–296.
- [8] A. C.-C. YAO, *Lower bounds for algebraic computation trees with integer inputs*, SIAM J. Comput., 20 (1991), pp. 655–668.