Discrete and continuous min-energy schedules for variable voltage processors

Minming Li[†], Andrew C. Yao[‡§], and Frances F. Yao[¶]

[†]Department of Computer Sciences and Technology and [‡]Center for Advanced Study, Tsinghua University, Beijing 100084, China; and [¶]Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong Special Administrative Region, China

Contributed by Andrew C. Yao, December 23, 2005

Current dynamic voltage scaling techniques allow the speed of processors to be set dynamically to save energy consumption, which is a major concern in microprocessor design. A theoretical model for min-energy job scheduling was first proposed a decade ago, and it was shown that for any convex energy function, the min-energy schedule for a set of *n* jobs has a unique characterization and is computable in $O(n^3)$ time. This algorithm has remained as the most efficient known despite many investigations of this model. In this work, we give an algorithm with running time $O(n^2 \log n)$ for finding the min-energy schedule. In contrast to the previous algorithm, which outputs optimal speed levels from high to low iteratively, our algorithm is based on finding successive approximations to the optimal schedule. At the core of the approximation is an efficient partitioning of the job set into high and low speed subsets by any speed threshold, without computing the exact speed function.

scheduling | energy efficiency | dynamic voltage scaling | optimization

Advances in processor, memory, and communication technologies have contributed to the tremendous growth of portable electronic devices. Because such devices are typically powered by batteries, energy efficiency has become a critical issue. An important strategy to achieve energy saving is through "dynamic voltage scaling" (DVS) (or "speed scaling"), which enables a processor to operate at a range of voltages and frequencies. Because energy consumption is at least a quadratic function of the supply voltage (hence CPU frequency/speed), it saves energy to execute jobs as slowly as possible while still satisfying all timing constraints. The associated scheduling problem is referred to as min-energy DVS scheduling.

A theoretical study of speed scaling scheduling was initiated by Yao, Demers, and Shenker (1). They formulated the DVS scheduling problem and gave an $O(n^3)$ algorithm for computing the optimal schedule.[‖] No special restriction was put on the power consumption function except convexity. To achieve optimality, it is assumed the processor speed may be set at any real value. This model will be referred to as the "continuous" model.

In practice, variable voltage processors can run at only a finite number of preset speed levels, although that number is increasing fast. (For example, the new Foxon technology will soon enable Intel server chips to run at 64 speed grades.) One can capture the discrete nature of the speed scale with a corresponding "discrete" scheduling model. It was observed in ref. 2 that an optimal discrete schedule for a job set can be obtained simply as follows: (*i*) construct the optimal continuous schedule, and (*ii*) individually adjust the "ideal" speed of each job by mapping it to the nearest higher and lower speed levels. The complexity of such an algorithm is thus the same as the continuous algorithm. Recently, it was shown in ref. 3 that the first step could be bypassed in a more efficient $O(dn \log n)$ algorithm where d is the number of speed levels. The algorithm works by directly partitioning the job set into two subsets (referred to as a bipartition), those requiring speed $\geq s$ and $< s$, respectively, for any specific speed level s.

In this work we present improved algorithms for both the continuous and the discrete DVS scheduling problems. We first derive a sharper characterization of job set bipartition than that given in ref. 3, which leads to an effective $O(n \log n)$ partitioning algorithm. Although the time complexity is the same as that achieved in ref. 3, the new partitioning algorithm is much simpler to implement. We then use it to construct the continuous optimal schedule through successive approximations in an $O(n^2 \log n)$ algorithm. It is an improvement over the longstanding $O(n^3)$ bound for the above problem.

Prior work directly related to the present work includes those papers cited above and an efficient algorithm for the (continuous) optimal schedule when job sets are structured as trees (4). On-line algorithms have been studied in refs. 1, 4, and 5. For an up-to-date survey on research in power/temperature management, see ref. 6.

The remainder of the work is organized as follows. We give the problem formulation and review some basic properties of the optimal continuous schedule in *Background*. *Bipartition of Jobs by Speed Threshold* describes an $O(n \log n)$ algorithm for job partitioning by any speed level, which forms the core of our scheduling algorithm. We then apply the partitioning algorithm to construct optimal schedules in *Scheduling Algorithms*. Some concluding remarks are given in *Conclusion*.

Background

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. We also refer to $[a_k, b_k] \subseteq [0, 1]$ as the interval of j_k and assume without loss of generality that $a_k < b_k$, and $\cup_k [a_k, b_k] = [0, 1]$ (or J spans $[0, 1]$). A schedule S for J is a pair of functions $(s(t), job(t))$ which defines, respectively, the processor speed and the job being executed at time t. Both functions are assumed to be piecewise continuous 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 the "DVS scheduling" (or sometimes "continuous DVS scheduling" to distinguish it from the discrete version below).

In the discrete version of the problem, we assume that the processor can run at d clock speeds $s_1 > s_2 > \cdots > s_d$. 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

Conflict of interest statement: No conflicts declared.

Abbreviations: DVS, dynamic voltage scaling; EDF, earliest deadline first.

[§]To whom correspondence should be addressed. E-mail: andrewcyao@tsinghua.edu.cn.

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

Algorithm 1. Basic Optimal Voltage Schedule (BOVS)

Input: job set J
Output: optimal voltage schedule S for J
  repeat
    Select $I^* = [z, z']$ with $g(I^*) = \max g(I)$
    Schedule $j_i \in J_{I^*}$ at $g(I^*)$ over I^* by EDF 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 a_k similarly
    end for
  until J is empty

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 using 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 details are given below.

Definition 1: For any interval $I \subseteq [0, 1]$, denote by J_I the subset of all jobs in J whose intervals are completely contained in I. The intensity of an interval I is defined to be

$$g(I) = \sum_{j_k \in J_I} R_k / |I|.$$

An interval I^* achieving maximum $g(I)$ over all possible intervals I defines a critical interval for the current job set. It is not hard to argue that the subset of jobs J_{I^*} can be feasibly scheduled at speed $g(I^*)$ over I^* by the earliest deadline first (EDF) principle. That is, at any time t, a job that is available for execution and having earliest deadline will be executed during $[t, t + \varepsilon]$. (Among jobs with the same deadline, the tie is broken by some fixed rule, say by the ordering of job indices. We refer to the resulting linear order as EDF order.) The interval I^* is then removed from $[0, 1]$; all remaining job intervals $[a_k, b_k]$ are updated to reflect the removal, and the algorithm recurses. The complete algorithm is give in Algorithm 1. We note that the optimal speed function s_{opt} for a job set is in fact unique.

Let $CI(i) \subseteq [0, 1]$ denote the ith critical interval of J, and $J_{CI(i)}$ the set of jobs executed during $CI(i)$. The following lemma is a direct consequence of the way critical intervals are successively selected.

Lemma 1. *A job $j_k \in J$ belongs to $\cup_{i=1}^{m} J_{CI(i)}$ if and only if its interval satisfies $[a_k, b_k] \subseteq \cup_{i=1}^{m} CI(i)$.*

A special tool, called s-schedules, was introduced in ref. 3 that can provide useful information regarding the optimal speed function for J without explicitly computing it. Our algorithms will also make use of s-schedules. For easy reference, we give the relevant definitions and properties below.

Definition 2: For any constant s, the s-schedule for J is an EDF schedule that uses constant speed s in executing any jobs of J. In general, s-schedules may have idle periods or unfinished jobs.

Definition 3: In a 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 union of all execution intervals for j_k with respect to S. Execution intervals with respect to the s-schedule will be called s-execution intervals.

It is easy to see that the s-schedule for n jobs will contain at most $2n$ s-execution intervals, because the end of each execution interval (including an idle interval) corresponds to the moment when either a job is finished or a new job arrives. Also, the s-schedule can be computed in $O(n \log n)$ time by using a priority queue to keep track of all jobs currently available, prioritized by deadlines.

The next lemma says that monotone relations between two speed functions for a job set J can induce certain monotone relations between the corresponding EDF schedules. These monotone properties will be useful when we study partitions of a job set by some speed threshold in the next section.

Definition 4: Let S_1 and S_2 be two EDF schedules for J with speed functions $s_1(t)$ and $s_2(t)$, respectively. We say S_1 dominates S_2 if $s_1(t) \geq s_2(t)$ for all t whenever S_1 is not idle. We say S_1 strictly dominates S_2 if $s_1(t) > s_2(t)$ for all t whenever S_1 is not idle.

Lemma 2 (3). *Let $J = \{j_1, \ldots, j_n\}$ by EDF ordering. Suppose S_1 and S_2 are two EDF schedules for J such that S_1 dominates S_2.*

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. *Suppose job j_k is finished at time t_0 under S_2. Then under S_1, job j_k will be finished no later than t_0 and, if the dominance is strict, be finished strictly earlier than t_0.*
4. *If S_2 is a feasible schedule for J, then so is S_1.*

Bipartition of Jobs by Speed Threshold

We describe a procedure that for any given speed threshold s, can properly separate J into two subsets: those jobs using speeds higher than s and those jobs using speeds lower than s, respectively, in the optimal schedule. This procedure forms the core of our min-energy scheduling algorithms. The basic ideas of such a partition and a corresponding algorithm were give in ref. 3. Here we will derive stronger characterizations, which then lead to a simpler algorithm.

Definition 5: Given a job set J and any constant s, let $J^{\geq s}$ and $J^{<s}$ denote the subsets 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 execution speed $\geq s$. By *Lemma 1*, a job j_k is in $J^{\geq s}$ if and only if its interval $[a_k, b_k] \subseteq T^{\geq s}$. Thus, $J^{\geq s}$ is uniquely determined by $T^{\geq s}$. We refer to $\langle T^{\geq s}, T^{<s} \rangle$ where $T^{<s} = [0, 1] - T^{\geq s}$ as the s-partition of J by time.

An example of J with nine jobs is given in Fig. 1 together with its optimal speed function $S_{\text{opt}}(t)$. The portion of $S_{\text{opt}}(t)$ lying above or exactly on 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.

We compute the partition $\langle T^{\geq s}, T^{<s} \rangle$ by finding the individual connected components of $T^{\geq s}$ and $T^{<s}$, respectively. Label the connected components of $T^{\geq s}$ in right-to-left order as $\langle T_1^{\geq s}, T_2^{\geq s}, \ldots \rangle$, and label those of $T^{<s}$ similarly as $\langle T_1^{<s}, T_2^{<s}, \ldots \rangle$. The cardinalities of these two sets differ by at most one. For ease of notation, we make their cardinalities equal by creating, if necessary, an empty rightmost component for $T^{\geq s}$ of the form $[1, 1]$, and/or an empty leftmost component for $T^{<s}$ of the form $[0, 0]$. Hence, we can represent these connected components with a sequence of $2p + 1$ numbers of the form $\langle 0 = B_{p+1} \leq A_p < B_p < \cdots < A_1 \leq B_1 = 1 \rangle$, such that $T_i^{\geq s} = [A_i, B_i]$ and $T_i^{<s} = [B_{i+1}, A_i]$, for $1 \leq i \leq p$.

The following lemma allows us to view the s-schedule for J as

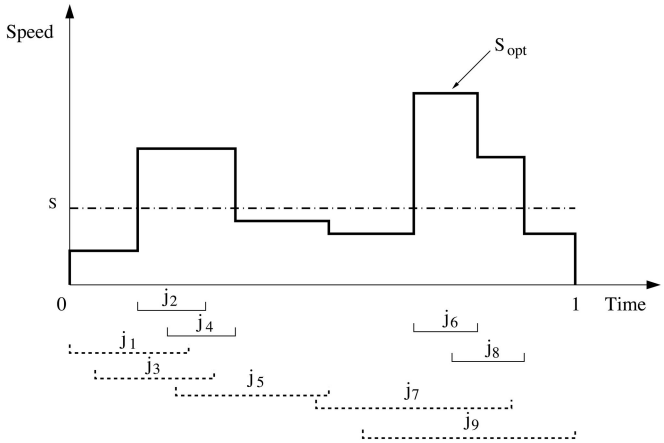

Fig. 1. The *s*-partition for a sample *J*. The jobs are represented by their intervals only and sorted according to deadline. Solid intervals represent jobs belonging to $J^{\geq s}$, while dashed intervals represent jobs belonging to $J^{<s}$.

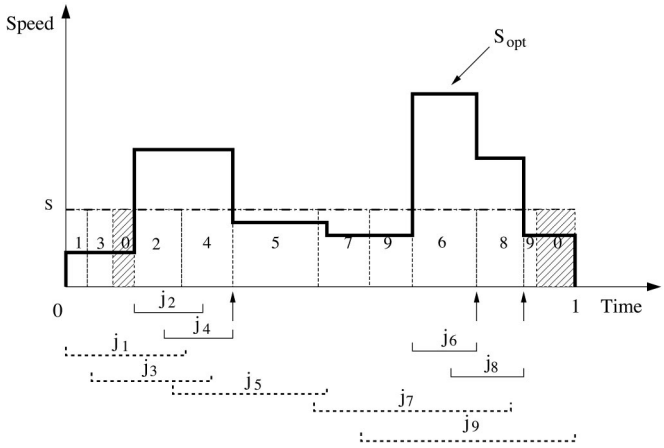

Fig. 2. The *s*-execution intervals for the same *J* in Fig. 1 are illustrated. Index *k* indicates that job j_k is being executed, whereas index 0 indicates a gap (idle time). Arrows point to tight deadlines.

composed of two separate *s*-schedules: one for job set $J^{<s}$ over $T^{<s}$, and another for job set $J^{<s}$ over $T^{<s}$. In comparing the *s*-schedule with S_{opt}, we can thus compare them over two disjoint regions for two disjoint job sets.

Lemma 3. In the *s-schedule for J*,

1. *all jobs executed during* $T^{<s}$ *belong to* $J^{<s}$,
2. *all jobs executed during* $T^{\geq s}$ *belong to* $J^{\geq s}$.

Proof: Statement 1 is obvious because all jobs belonging to $J^{\geq s}$ have intervals disjoint from $T^{<s}$, hence, will not be executed during $T^{<s}$. To prove statement 2, we consider any component $T_i^{\geq s} = [A_i, B_i]$ of $T^{\geq s}$. All jobs j_k of $J^{<s}$ with deadlines $b_k \leq B_i$ will have been finished by time A_i under S_{opt}; thus by *Lemma 2* they also will have been finished by time A_i under the *s*-schedule. [Note that statement 1 allows us to compare the *s*-schedule with S_{opt} for the same job set $J^{<s}$.] Therefore, such jobs will not be executed during $T_i^{\geq s}$. The same is also true for jobs of $J^{<s}$ with deadlines $>B_i$. The latter claim can be proved by comparing the *s*-schedule with S_{opt} for the job set $J_i^{\geq s}$ consisting of jobs whose intervals are contained in $T_i^{\geq s}$. Note that schedule S_{opt} executes $J_i^{\geq s}$ without idle time during $T_i^{\geq s}$. It follows from *Lemma 2* that, during $T_i^{\geq s}$ the *s*-schedule also will execute $J_i^{\geq s}$ without idle time and hence will not execute any jobs with deadlines of $>B_i$. We have proved the lemma.

We construct the *s*-partition by inductively finding the rightmost pair of components $\{T_1^{\geq s}, T_1^{<s}\}$, remove them from $[0, 1]$, and then repeat the process. It identifies $T_1^{\geq s} = [A_1, B_1]$ and $T_1^{<s} = [B_2, A_1]$ by locating their boundary points B_2, A_1, and B_1 through some special characteristics that we discuss in the following.

Definition 6: In the *s*-schedule for *J*, we say a job 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 execution interval in the *s*-schedule is called a gap. Note that a gap must be of the form $[t, a]$ where $t < a$ and t corresponds to the end of the final execution block of some job, while a corresponds to a job arrival or $a = 1$. We also include a special execution interval $[0, 0]$ at the beginning of the *s*-schedule and regard it as a gap.

Fig. 2 depicts the *s*-schedule for the sample job set *J* given in Fig. 1. An *s*-execution interval indexed by *k* indicates that job j_k is being executed, except when $k = 0$, which indicates a gap (idle interval). The tight deadlines are marked by arrows. By examining the *s*-partition of time $\langle T^{\geq s}, T^{<s}\rangle$ for *J*, we notice that (*i*) tight deadlines exist only in $T^{\geq s}$, and (*ii*) each connected component of $T^{\geq s}$ ends with a tight deadline. The following lemma from ref. 3 states that these properties always hold for any job set.

Lemma 4 (3).

1. *Tight deadlines do not exist in* $T^{<s}$.
2. *The right endpoint* B_i *of* $T_i^{\geq s} = [A_i, B_i]$ *must be a tight deadline for* $2 \leq i \leq p$.

Definition 7: Given a gap $[t, a]$ in an *s*-schedule, we define the expansion of $[t, a]$ to be the smallest interval $[b, a] \supseteq [t, a]$ where b is a tight deadline (see Fig. 3). To ensure that the expansion of a gap always exists, we adopt the convention that 0 is considered a tight deadline. In particular, the expansion of the special gap $[0, 0]$ is $[0, 0]$ itself.

Lemma 5.

1. *Gaps do not exist within* $T_1^{\geq s}$.
2. $T_1^{<s} = [B_2, A_1]$ *must end with a gap.*
3. $T_1^{<s} = [B_2, A_1]$ *corresponds to the expansion of the rightmost gap in the s*-schedule.

Proof: Property 1 was already established in the proof of *Lemma 3*. For property 2, we can compare the *s*-schedule with

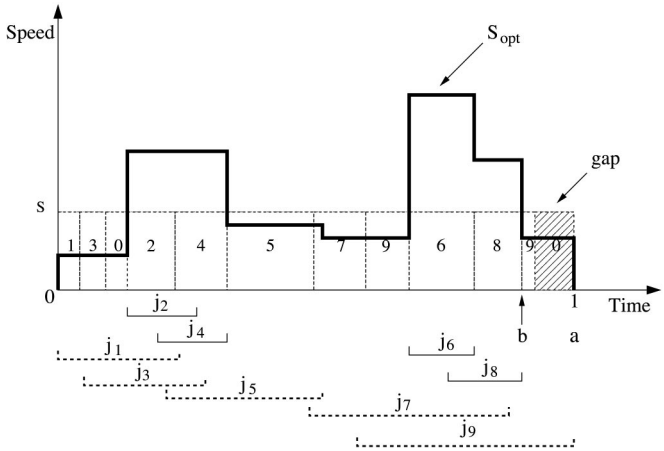

Fig. 3. Gap expansion. The indicated gap will be expanded into $[b, a]$, a connected component of $T^{\leq s}$.

S_{opt} over $T^{<s}$ by considering only those jobs in $J^{<s}$, based on *Lemma 3*. Because the s-schedule strictly dominates S_{opt} over $T^{<s}$, *Lemma 2* implies that the last job executed by the s-schedule within $T_1^{<s}$ must be finished strictly earlier than by S_{opt}. Hence, there must be a gap at the end of $T_1^{<s}$. Property 3 then follows from property 2 and the second property of *Lemma 4*.

Note that *Lemma 5* asserts the existence of gaps in $T_1^{<s}$, but not in $T_i^{<s}$, for $i \geq 2$. It is because the s-schedule may keep itself fully occupied during $T_i^{<s}$ for $i \geq 2$ by executing jobs of $J^{<s}$ ahead of the S_{opt} schedule. Thus, in the initial s-schedule, only the first component $T_1^{<s}$ of $T^{<s}$ is sure to contain gaps.

Lemma 6 below provides the basis for an inductive approach to construct the s-partition by finding one pair of connected components $\{T_i^{\geq s}, T_i^{<s}\}$ at a time, then deleting certain associated subsets of jobs (defined below) $\{J_i^{\geq s}, J_i^{<s}\}$ from J. This step is repeated for $1 \leq i \leq p$. *Lemma 7* shows that the required updates to the s-schedule (the main data structure used by the algorithm) to reflect the removal of jobs are quite straightforward.

Definition 8: Let $J_i^{\geq s}$ denote the subset of all jobs in J whose intervals are completely contained in $T_i^{\geq s}$. Let $J_i^{<s}$ denote the subset of all jobs in J whose intervals have nonempty intersections with $T_i^{<s}$ but have empty intersections with each of $\{T_1^{<s}, \ldots, T_{i-1}^{<s}\}$.

Lemma 6. *Let $J' = J - J_1^{\geq s} - J_1^{<s}$. The s-partition of job set J' is consistent with the s-partition of job set J. That is, a job in J' has speed $< s$ in the optimal schedule for J' if and only it has speed $< s$ in the optimal schedule for J.*

Proof: We claim that (*i*) the optimal speed function of J' coincides with the optimal speed function of J for all jobs in $J_2^{\geq s}, \ldots, J_{i-1}^{\geq s}$, and (*ii*) all jobs in $J^{<s} - J_1^{<s}$ have speeds $< s$ in the optimal speed function of J'. First, the deletion from J of all jobs in $J_1^{\geq s}$ (whose intervals are contained in $T_1^{\geq s}$) will not affect the optimal speed for any job in the remaining $J_i^{\geq s}$ (whose intervals are contained in $T_i^{\geq s}$ which is disjoint from $T_1^{\geq s}$), by the way the intensity function $g(I)$ is defined. Second, the deletion of jobs in $J_1^{<s}$ clearly will not change the optimal speed for any job in $J^{\geq s}$. Thus, (*i*) is true. To prove (*ii*), consider the speeds m_1 and m_2, defined as the highest speeds used by any job of $J^{<s} - J_1^{<s}$ in the optimal schedules for J and for J', respectively. By definition, $m_1 < s$. Then, by examining the way the highest intensity $g(I^*)$ is selected for jobs in $J^{<s} - J_1^{<s}$, and using similar arguments as for (*i*), it is easy to see that $m_2 \leq m_1 < s$. We have proved (*ii*) and hence the lemma.

Lemma 7. *Let $J = \{j_1, \ldots, j_n\}$ by EDF ordering. For any $m < n$, the s-schedule for the job set $J' = \{j_1, \ldots, j_m\}$ can be obtained from the s-schedule of J by simply changing the execution intervals of each job in $\{j_{m+1}, \ldots, j_n\}$ into idle intervals (i.e., gaps).*

The above lemma is easy to prove by induction on m. Finally, the s-partition can be obtained by combining the subsets that have been identified.

Lemma 8.

1. $J^{\geq s} = \cup_{i=1}^{p} J_i^{\geq s}$.
2. $J^{<s} = \cup_{i=1}^{p} J_i^{<s}$.

Proof: By *Lemma 1*, a job $j_k \in J^{\geq s}$ if and only if its interval $[a_k, b_k] \subseteq T^{\geq s}$, or equivalently, if and only if $[a_k, b_k] \subseteq T_i^{\geq s}$ for one of the connected components $T_i^{\geq s}$ of $T^{\geq s}$. This fact proves property 1. For property 2, note that $j_k \in J^{<s}$ if and only if its interval $[a_k, b_k] \cap T^{<s} \neq \emptyset$, hence if and only if $j_k \in J_i^{<s}$ for some i, $1 \leq i \leq p$.

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

Algorithm 2. Bipartition

Input: speed s, and job set J
Output: s-partition $\langle J^{\geq s}, J^{<s}\rangle$ of J
  Sort jobs into $j_1, \ldots, j_n$ by EDF ordering
  Generate the s-schedule for J
  $i \leftarrow 0$
  $M \leftarrow n$
  $b_0 \leftarrow 0$
  $B_1 \leftarrow 1$
  repeat
    $i \leftarrow i + 1$
    take the rightmost gap $[t, a]$ and find the expansion $[b, a]$ of $[t, a]$
    $A_i \leftarrow a$ (this defines component $T_i^{\geq s} = [A_i, B_i]$)
    $B_{i+1} \leftarrow b$ (this defines component $T_i^{<s} = [B_{i+1}, A_i]$)
    while $b_M > A_i$ **do**
    if $a_M \geq A_i$ **then**
      add j_M to $J_i^{\geq s}$
      else
        add j_M to $J_i^{<s}$
      end if
      remove j_M from J
      $M \leftarrow M - 1$
    end while
    while $b_M > B_{i+1}$ **do**
      add j_M to $J_i^{<s}$
      remove j_M from J
      $M \leftarrow M - 1$
    end while
  until $M = 0$
  $J^{\geq s} \leftarrow J_1^{\geq s} \cup \cdots \cup J_i^{\geq s}$
  $J^{<s} \leftarrow J_1^{<s} \cup \cdots \cup J_i^{<s}$
  Return $\langle J^{\geq s}, J^{<s}\rangle$

Theorem 1. *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 algorithm uses the characterizations given in *Lemmas 4* and *5* to locate the boundary points B_{i+1}, A_i, and B_i for components $T_i^{\geq s}$ and $T_i^{<s}$. It then identifies the jobs belonging to $J_i^{\geq s}$ or to $J_i^{<s}$, and removes them from J. *Lemma 6* guarantees that the above process can be carried out inductively to find all components $T_i^{\geq s}$ and $T_i^{<s}$. The desired s-partition is obtained by taking unions according to *Lemma 8*. We now analyze the time complexity of the algorithm. Generating the initial s-schedule takes $O(n \log n)$ time. The remaining computation can be done in $O(n)$ time with appropriate data structures. One can use a linked list to represent the s-schedule, and linked sublists to represent the s-execution intervals for every job. Each execution interval and each job interval are examined only a constant number of times, because all pointers used in the algorithm make a single pass from right to left. Therefore, the total running time of the algorithm is $O(n \log n)$.

Scheduling Algorithms

We now apply Algorithm Bipartition to the computation of optimal schedules. We will discuss the continuous case and the discrete case separately in the following two subsections.

Continuous Case. For a job set J, define the "support" T of J to be the union of all job intervals in J. Define $avr(J)$, the "average rate" of J to be the total workload of J divided by $|T|$. We will use $avr(J)$ as the speed threshold to perform a bipartition on J, which, according to the next lemma, produces two nonempty subsets unless $S_{opt}(t)$ is constant for J.

Lemma 9. *Let $s = avr(J)$. Then $T^{\geq s} \neq \emptyset$ if $J \neq \emptyset$. Furthermore, the following three conditions are equivalent.*

Algorithm 3. Partitioned Optimal Voltage Schedule (POVS)

Input: job set J
Output: (Continuous) optimal voltage schedule S_{opt} for J
  if $J = \emptyset$ **then**
    return
  end if
  $s \leftarrow avr(J)$
  $\langle J^{\geq s}, J^{<s} \rangle \leftarrow$ *Bipartition* (J, s)
  if $T^{<s} = \emptyset$ **then**
    return the s-schedule over T
  else
    return the union of schedules $POVS(J^{\geq s}, T^{\geq s})$ and $POVS(J^{<s}, T^{<s})$
  end if

1. $T^{<s} = \emptyset$.
2. $T^{\geq s} = T$.
3. $S_{\text{opt}}(J) = s$ *over* T.

Proof: Conditions 1 and 2 are obviously equivalent as $T^{\geq s} \cup T^{<s} = T$. Since $\int_T S_{\text{opt}}\, dt = \int_T s\, dt = \Sigma R_k$, we have $\int_{T^{\geq s}} (S_{\text{opt}} - s)dt = \int_{T^{<s}} (s - S_{\text{opt}})dt$. If condition 1 is true then $\int_{T^{<s}} (s - S_{\text{opt}})dt = 0$, which implies $\int_{T^{\geq s}} (S_{\text{opt}} - s)dt = 0$, hence $S_{\text{opt}}(J) = s$ over $T^{\geq s}$ $(= T)$ and condition 3 is true. Conversely, condition 3 implies condition 1 by the definition of $T^{<s}$. We have proved that conditions 1 and 3 are equivalent.

Theorem 2. *Algorithm* 3 *computes a (continuous) optimal voltage schedule for a job set* J *in* $O(n^2 \log n)$ *time.*

Proof: The correctness of Algorithm 3 follows from *Theorem 1* and *Lemma 9*. The process of repeated partitions can be represented by a binary tree where each internal node v corresponds to a bipartition. After initially sorting the job arrivals and deadlines, the cost of the bipartition at each node v is $O(n \log n)$ in the size of the subtree at v by *Theorem 1*. The sum over all internal nodes is $O(P \log n)$ where P is the total path lengths of the tree and is at most $O(n^2)$. Hence, the time complexity of the algorithm is $O(n^2 \log n)$.

Discrete Case. By applying Algorithm 2 repeatedly, one can partition J into d subsets corresponding to d speed levels in time $O(dn \log n)$. We then schedule the jobs in each subset J_i with speed levels s_i and s_{i+1} by applying a two-level scheduling algorithm given in ref. 3. The latter algorithm, when given a set J of n jobs and two speed levels $s > s'$ that are known to satisfy $s > s_{\text{opt}}(t) \geq s'$ for all t, can compute the optimal schedule for J with discrete speed levels s and s' in $O(n \log n)$ time. We incorporate these two steps in a single loop as shown in Algorithm 4. Algorithm 4 is simpler than the discrete scheduling algorithm given in ref. 3, although the time complexity $O(dn \log n)$ is the same. We also remark that an $\Omega(n \log n)$ lower bound in the algebraic decision tree model was proven in ref. 3 for the discrete DVS scheduling problem. Hence, Algorithm 4 has optimal complexity if d is considered a fixed constant.

Algorithm 4. Discrete Optimal Voltage Schedule (DOVS)

Input:
job set J
speed levels: $s_1 > s_2 > \ldots > s_d > s_{d+1} = 0$
Output:
Discrete Optimal Voltage Schedule for J
  for $i = 1$ to d **do**
    Obtain $J^{\geq s_{i+1}}$ from J using Algorithm 2
    $J_i \leftarrow J^{\geq s_{i+1}}$
    Schedule jobs in J_i using two-level scheduling algorithm given in ref. 3 with speeds s_i and s_{i+1}
    $J \leftarrow J - J_i$
    Update J as in Algorithm 1
  end for
  The union of the schedules gives the optimal Discrete DVS schedule for J

Conclusion

In this work we considered the problem of job scheduling on a variable voltage processor to minimize overall energy consumption. For the continuous case where the processor can run at any speed, we give a min-energy scheduling algorithm with time complexity $O(n^2 \log n)$. This result improves over the best previous bound of $O(n^3)$. For the discrete case with d preset speed levels, we obtain a simpler algorithm than that given in ref. 3, with the same time complexity $O(dn \log n)$. The basis of both algorithms is an efficient method to partition a job set, by any speed level, into high-speed and low-speed subsets. This strategy, quite natural for the discrete problem, turned out to be also effective for the continuous case by enabling successive approximations to the optimum. Our results may provide some insights into the min-energy scheduling problem. They also should be useful in generating optimal schedules as benchmarks for evaluating heuristic algorithms. We propose as an open problem to investigate whether the $O(n^2 \log n)$ time complexity could be further improved.

This work was supported in part by Research Grants Council of Hong Kong Grant CityU 122105; National Natural Science Foundation of China Grant 60135010, 60321002, and 60553001; and Chinese National Key Foundation Research and Development Plan Grant 2004CB318108.

1. Yao, F., Demers, A. & Shenker, S. (1995) in *Proceedings of the 36th IEEE Conference on the Foundations of Computer Science (FOCS)* (IEEE, New York), pp. 374–382.
2. Kwon, W. & Kim, T. (2005) *ACM Trans. Embedded Computing Systems* **4,** 211–230.
3. Li, M. & Yao, F. F. (2005) *SIAM J. Computing* **35,** 658–671.
4. Li, M., Liu, J. B. & Yao, F. F. (2005) in *Proceedings of the Eleventh International Computing and Combinatorics Conference*, ed. Wang, L. (Springer, Berlin), pp. 283–296.
5. Bansal, N., Kimbrel, T. & Pruhs, K. (2004) in *Proceedings of the 45th Annual IEEE Conference on the Foundations of Computer Science (FOCS)* (IEEE, New York), pp. 520–529.
6. Irani, S. & Pruhs, K. (2005) *ACM SIGACT News*, **36,** 63–76.