An $O(n^2)$ Algorithm for Computing Optimal Continuous Voltage Schedules

Minming Li · **Frances F. Yao** · **Hao Yuan**

Abstract Dynamic Voltage Scaling techniques allow the processor to set its speed dynamically in order to reduce energy consumption. In the continuous model, the processor can run at any speed, while in the discrete model, the processor can only run at finite number of speeds given as input. The current best algorithm for computing the optimal schedules for the continuous model runs at $O(n^2 \log n)$ time for scheduling n jobs. In this paper, we improve the running time to $O(n^2)$ by speeding up the calculation of s-schedules using a more refined data structure. For the discrete model, we improve the computation of the optimal schedule from the current best $O(dn \log n)$ to $O(n \log \max\{d, n\})$ where d is the number of allowed speeds.

1 Introduction

Energy efficiency is always a primary concern for chip designers not only for the sake of prolonging the lifetime of batteries which are the major power supply of portable electronic devices but also for the environmental protection purpose when large facilities like data centers are involved. Currently, processors capable of operating at a range of frequencies are already available, such as Intel's SpeedStep technology and AMD's PowerNow technology. The capability of the processor to change voltages is often referred to in the literature as DVS (Dynamic Voltage Scaling) techniques. For DVS processors, since energy

Department of Computer Science, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong. Email: `minmli@cs.cityu.edu.hk`

Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China. Email: `csfyao@cityu.edu.hk`

Department of Computer Science, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong. Email: `haoyuan@cityu.edu.hk`

Address(es) of author(s) should be given

consumption is at least a quadratic function of the supply voltage (which is proportional to CPU speed), it saves energy to let the processor run at the lowest possible speed while still satisfying all the timing constraints, rather than running at full speed and then switching to idle.

One of the earliest theoretical models for DVS was introduced by Yao, Demers and Shenker [27] in 1995. They assumed that the processor can run at any speed and each job has an arrival time and a deadline. They gave a characterization of the minimum-energy schedule (MES) and an $O(n^3)$ algorithm for computing it which is later improved to $O(n^2 \log n)$ by [21]. No special assumption was made on the power consumption function except convexity. Several online heuristics were also considered including the Average Rate Heuristic (AVR) and Optimal Available Heuristic (OPA). Under the common assumption of power function $P(s) = s^\alpha$, they showed that AVR has a competitive ratio of $2^{\alpha-1}\alpha^\alpha$ for all job sets. Thus its energy consumption is at most a constant times the minimum required. Later on, under various related models and assumptions, more algorithms for energy-efficient scheduling have been proposed.

Bansal et al. [7] further investigated the online heuristics for the model proposed by [27] and proved that the heuristic OPA has a tight competitive ratio of α^α for all job sets. For the temperature model where the temperature of the processor is not allowed to exceed a certain thermal threshold, they showed how to solve it within any error bound in polynomial time. Recently, Bansal et al. [5] showed that the competitive analysis of AVR heuristic given in [27] is essentially tight. Quan and Hu [24] considered scheduling jobs with fixed priorities and characterized the optimal schedule through transformations to MES [27]. Yun and Kim [28] later on showed the NP-hardness to compute the optimal schedule.

Pruhs et al. [23] studied the problem of minimizing the average flow time of a sequence of jobs when a fixed amount of energy is available and gave a polynomial time offline algorithm for unit-size jobs. Bunde [9] extended this problem to the multiprocessor scenario and gave some nice results for unit-size jobs. Chan et al. [10] investigated a slightly more realistic model where the maximum speed is bounded. They proposed an online algorithm which is $O(1)$-competitive in both energy consumption and throughput. More work on the speed bounded model can be found in [6] [11] [18].

Ishihara and Yasuura [16] initiated the research on discrete DVS problem where a CPU can only run at a set of given speeds. They solved the case when the processor is only allowed to run at two different speeds. Kwon and Kim [17] extended it to the general discrete DVS model where the processor is allowed to run at speeds chosen from a finite speed set. They gave an $O(n^3)$ algorithm for this problem based on the MES algorithm in [27], which is later improved in [19] to $O(dn \log n)$ where d is the allowed number of speeds.

When the CPU can only change speed gradually instead of instantly, [13] discussed about some special cases that can be solved optimally in polynomial time. Later, Wu et al. [26] extended the polynomial solvability to jobs with agreeable deadlines. Irani et al. [14] investigated an extended scenario where

the processor can be put into a low-power sleep state when idle. A certain amount of energy is needed when the processor changes from the sleep state to the active state. The technique of switching processors from idle to sleep and back to idle is called Dynamic Power Management (DPM) which is the other major technique for energy efficiency. They gave an offline algorithm that achieves 2-approximation and online algorithms with constant competitive ratios. Recently, Albers and Antoniadis [3] proved the NP-hardness of the above problem and also showed some lower bounds of the approximation ratio. Pruhs et al. [22] introduced profit into DVS scheduling. They assume that the profit obtained from a job is a function on its finishing time and on the other hand money needs to be paid to buy energy to execute jobs. They give a lower bound on how good an online algorithm can be and also give a constant competitive ratio online algorithm in the resource augmentation setting. A survey on algorithmic problems in power management for DVS by Irani and Pruhs can be found in [15]. Most recent surveys by Albers can be found in [1] [2].

In [20], the authors showed that the optimal schedule for tree structured jobs can be computed in $O(n^2)$ time. In this paper, we prove that the optimal schedule for general jobs can also be computed in $O(n^2)$ time, improving upon the previously best known $O(n^2 \log n)$ result [21]. The remaining paper is organized as follows. Section 2 will give the problem formulation. Section 3 will discuss the linear implementation of an important tool — the s-schedule used in the algorithm in [21]. Then we use the linear implementation to improve the calculation of the optimal schedule in Section 4. In Section 5, we give improvements in the computation complexity of the optimal schedule for the discrete model. Finally, we conclude the paper in Section 6.

2 Models and Preliminaries

We consider the single processor setting. A job set $J = \{j_1, j_2, \ldots, j_n\}$ over $[0, 1]$ is given where each job j_k is characterized by three parameters: arrival time a_k, deadline b_k, and workload R_k. Here workload means the required number of CPU cycles. We also refer to $[a_k, b_k] \subseteq [0, 1]$ as the interval of j_k. A *schedule* S for J is a pair of functions $(s(t), job(t))$ which defines the processor speed and the job being executed at time t respectively. Both functions are assumed to be piecewise continuous with finitely many discontinuities. A *feasible* schedule must give each job its required workload between its arrival time and deadline with perhaps intermittent execution. We assume that the power P, or energy consumed per unit time, is $P(s) = s^\alpha$ ($\alpha \geq 2$) where s is 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 feasibility scheduling problem is to find a feasible schedule that minimizes $E(S)$ for any given job set J. We refer to this problem as the continuous *DVS* scheduling problem.

For the continuous DVS scheduling problem, the optimal schedule S_{opt} is characterized by 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 MES in [27] 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 $I \subseteq [0, 1]$, we use J_I to denote the subset of 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 known 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 which is waiting to be executed and having earliest deadline will be executed during $[t, t + \epsilon]$. The interval I^* is then removed from $[0, 1]$; all the remaining job intervals $[a_k, b_k]$ are updated to reflect the removal, and the algorithm recurses. We denote the optimal schedule which guarantees feasibility and consumes minimum energy in the continuous DVS model as OPT.

The authors in [21] later observed that in fact the critical intervals do not need to be located one after another. Instead, one can use a concept called s-schedule defined below to do bipartition on jobs which gradually approaches the optimal speed curve.

Definition 2 For any constant s, the s-schedule for J is an EDF schedule which uses a constant speed s in executing any jobs of J. It will give up a job when the deadline of the job has passed. 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 contains at most $2n$ s-execution intervals, since 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 all jobs currently available, prioritized by their deadlines. In the next section, we will show that the s-schedule can be computed in linear time.

3 Computing an s-Schedule in Linear Time

In this work, we assume that the underlying computational model is the unit-cost RAM model with word size $\Theta(\log n)$. This model is assumed only for the purpose of using a special union-find algorithm by Gabow and Tarjan [12].

1 Initialize $e_i \leftarrow t_i$ for $1 \leq i < m$. ; *arrival time*
2 **for** *k=1* **to** n **do**   *b1 ≤ b2 ≤ ... ≤ bn*
3   Let i be the rank of a_k in T, i.e., $t_i = a_k$. ;
4   Initialize $r \leftarrow R_k$, where r denotes the remaining workload to be scheduled. ;
5   **while** $r > 0$ **do**
6   Search for an earliest *non-empty* canonical time interval $[e_p, t_{p+1})$ such that $e_p \geq t_i$. ;
7   **if** $e_p \geq b_k$ **then**
8   Break the while loop because the job cannot be finished. *miss deadline*
9   **end**
10   Set $u \leftarrow \min\{b_k, t_{p+1}\}$.   *[ep,u) u<= deadline*
11   **if** $r > s \cdot (u - e_p)$ **then**
12   Schedule job k at $[e_p, u)$. ;
13   Update $e_p \leftarrow u$. ;
14   Update $r \leftarrow r - s \cdot (u - e_p)$. ;   *Not Finished!*
15   **else**
16   Schedule job k at $[e_p, e_p + r/s)$. ;
17   Update $e_p \leftarrow e_p + r/s$. ;
18   Update $r \leftarrow 0$.   *Finished!*
19   **end**
20   **end**
21 **end**

Algorithm 1: Algorithm for Computing an s-Schedule

Theorem 1 *If for each k, the rank of a_k in $\{a_1, a_2, \ldots, a_n\}$ and the rank of b_k in $\{b_1, b_2, \ldots, b_n\}$ are pre-computed, then the s-schedule can be computed in linear time in the unit-cost RAM model.*

We make the following two assumptions:

- the jobs are already sorted according to their deadlines;
- for each job j_k, we know the rank of a_k in the arrival time set $\{a_1, a_2, \ldots, a_n\}$.

Because of the first assumption and without loss of generality, we assume that $b_1 \leq b_2 \leq \ldots \leq b_n$. Algorithm 1 schedules the jobs in the order of their deadlines. When scheduling job k, the algorithm tries to search for an earliest available time interval and schedule the job in it, and then repeat the process until all the workload of the job is scheduled or unable to find such a time interval before the deadline. A more detailed discussion of the algorithm is given below.

Let T be $\{a_1, a_2, \ldots, a_n, 1, 1 + \epsilon\}$. Note that the times "1" and "$1 + \epsilon$" (where ϵ is any fixed positive constant) are included in T for simplifying the presentation of the algorithm. Denote the size of T by m. Denote t_i to be the i-th smallest element in T. Note that the rank of any a_k in T is known. During the running of the algorithm, we will maintain the following data structure:

Definition 4 For each $1 \leq i < m$, the algorithm maintains a value e_i, whose value is in the range $[t_i, t_{i+1}]$. The meaning of e_i is that: the time interval $[t_i, e_i)$ is fully occupied by some jobs, and the time interval $[e_i, t_{i+1})$ is idle.

If $[t_i, t_{i+1})$ is fully occupied, then e_i is t_{i+1}. Note that such a time e_i always exists during the running of the algorithm, which will be shown later when
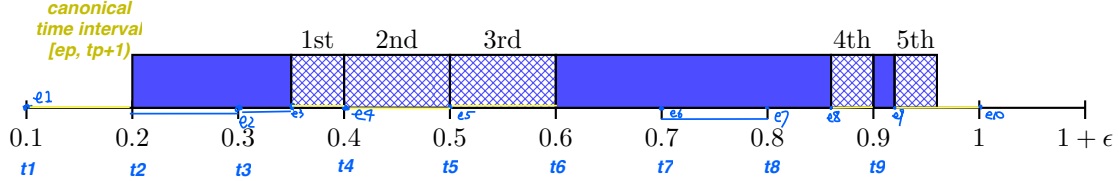
Fig. 1: An illustration for example 2.

we discuss how to maintain e_i. At the beginning of the algorithm, we assume that the processor is idle for the whole time period. That means $e_i = t_i$ for $1 \le i < m$ (see line 1 of Algorithm 1).

Example 1 An example for demonstrating the usage of the e_i data structure is given below: Assume that $T = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1 + \epsilon\}$. At some point during the execution of the algorithm, if some jobs have been scheduled to run at time intervals $[0.2, 0.35), [0.6, 0.86), [0.9, 0.92)$, then we will have $e_1 = 0.1$, $e_2 = 0.3$, $e_3 = 0.35$, $e_4 = 0.4$, $e_5 = 0.5$, $e_6 = 0.7$, $e_7 = 0.8$, $e_8 = 0.86$, $e_9 = 0.92$, and $e_{10} = 1$.

Before we analyze the algorithm, we need to define an important concept called canonical time interval.

Definition 5 During the running of the algorithm, a canonical time interval is a time interval of the form $[e_p, t_{p+1})$, where $1 \le p < m$. When $e_p = t_{p+1}$, we call it an empty canonical time interval.

Note that a non-empty canonical time interval is always idle based on the definition of e_p. Any arrival time a_k will not lie inside any canonical time interval but it is possible that a_k will touch any of the two ending points, i.e., for any $1 \le p < m$, we have either $a_k \le e_p$ or $a_k \ge t_{p+1}$. Therefore, if we want to search for a time interval to run a job at or after time a_k, then we should always look for the earliest non-empty canonical time interval $[e_p, t_{p+1})$ where $e_p \ge a_k$.

In Algorithm 1, a variable r is used to track the workload to be scheduled. Lines 5-20 try to schedule j_k as early as possible if $r > 0$. Line 6 tries to search

for an earliest non-empty canonical time interval $[e_p, t_{p+1})$ no earlier than the arrival time of j_k (i.e., $e_p \geq a_k$). Such a p always exists because there is always a non-empty canonical time interval $[1, 1 + \epsilon)$. Line 7-9 means that, if e_p is not earlier than the deadline of j_k, then the job cannot be finished. Line 10 sets a value of u, whose meaning is that $[e_p, u)$ can be used to schedule the job. The value of u is no later than the deadline of j_k. Lines 12-14 process the case when the remaining workload of j_k cannot be finished in the time interval $[e_p, u)$. Lines 16-18 process the case when the remaining workload of j_k can be finished in the time interval $[e_p, u)$. In the first case, line 13 updates e_p to u because the time interval $[t_p, u)$ is occupied and $[u, t_{p+1})$ is idle. In the second case, a time of r/s is occupied by j_k after the time e_p, so e_p is increased by r/s.

Example 2 Following the example provided in the previous example, assume that the speed is $s = 1$, if we are to schedule a job j_k, where $a_k = 0.3, b_k = 0.96, R_k = 0.35$, the algorithm will proceed as follows: At the beginning, r will be initialized to 0.35, and $i = 3$ (because $a_k = 0.3 = t_3$; see line 3). Line 6 will then get the interval $[e_3, t_4) = [0.35, 0.4)$ as an earliest non-empty canonical time interval, and a workload of $(0.4 - 0.35)s = 0.05$ is scheduled at that time interval. The values of e_3 will be updated to 0.4 accordingly. Now, r becomes $0.35 - 0.05 = 0.3$, and line 6 will get the time interval $[e_4, t_5) = [0.4, 0.5)$ to schedule the job. After that r becomes $0.3 - (0.5 - 0.4)s = 0.2$, and $e_4 = 0.5$. Line 6 then gets the time interval $[e_5, t_6) = [0.5, 0.6)$ to schedule the job, and r will be further reduced to 0.1. The values of e_5 will be updated to 0.6. The next time interval found will be $[e_8, t_9) = [0.86, 0.9)$, and r will become $0.1 - (0.9 - 0.86)s = 0.06$. The values of e_8 will be updated to 0.9. The remaining earliest non-empty canonical time interval is $[e_9, t_{10}) = [0.92, 1)$, but the deadline of the job is 0.96, so only $[0.92, 0.96)$ will be used to schedule the job, and r will be 0.02. The value of e_9 is then updated to 0.96. Finally, $[e_9, t_{10}) = [0.96, 1)$ is the remaining earliest non-empty canonical time interval, but $e_9 \geq b_k$, so line 7-9 will break the loop, and j_k will be an unfinished job. A graphical illustration is provided in Figure 1. The solid rectangles represent the time intervals occupied by some jobs before scheduling j_k. The cross-hatched rectangles represent the time intervals that are used to schedule j_k. The q-th cross-hatched rectangle (where $1 \leq q \leq 5$) is the q-th time interval scheduled according to this example. Note that all the cross-hatched rectangles except the 5-th one are canonical time intervals right before scheduling j_k.

The most critical part of the algorithm is Line 6, which can be implemented efficiently by the following folklore method using a special union-find algorithm developed by Gabow and Tarjan [12] (see also the discussion of the decremental marked ancestor problem [4]). At the beginning, there is a set $\{i\}$ for each $1 \leq i < m$. The name of a set is the largest element of the set. Whenever e_p is updated to t_{p+1} (i.e., there is not any idle time in the interval $[t_p, t_{p+1})$), we make a union of the set containing p and the set containing $p + 1$, and set the name of this set to be the name of the set containing $p + 1$. After the union, the two old sets are destroyed. In this way, a set is always an interval of

integers. For a set whose elements are $\{q, q+1, \ldots, p\}$, the semantic meaning is that, $[t_q, e_p)$ is fully scheduled but $[e_p, t_{p+1})$ is idle. Therefore, to search for an earliest non-empty canonical time interval beginning at or after time t_i, we can find the set containing i, and let p be the name of the set, then $[e_p, t_{p+1})$ is the required time interval.

Example 3 An example of the above union-find process for scheduling j_k in the previous example is given below: Before scheduling j_k, we have the sets $\{1\}, \{2,3\}, \{4\}, \{5\}, \{6,7,8\}, \{9\}, \{10\}$. The execution of line 6 will always try to search for a set that contains the element $i = 3$. Therefore, the first execution will find the set $\{2,3\}$, so p will be 3. After that, e_3 becomes $t_4 = 0.4$, so the algorithm needs to make a union of the sets $\{2,3\}$ and $\{4\}$ to get $\{2,3,4\}$. Similarly, the next execution will find the set $\{2,3,4\}$, so $p = 4$. The algorithm will then make a union of $\{2,3,4\}$ and $\{5\}$ to get $\{2,3,4,5\}$. For the next execution, the set $\{2,3,4,5\}$ will be found, and it will be merged with $\{6,7,8\}$ to get $\{2,3,4,5,6,7,8\}$. In this case, $p = 8$, and the earliest non-empty canonical time interval is $[e_p, t_{p+1}) = [0.86, 0.9)$. After e_8 is updated to $t_9 = 0.9$, the algorithm will merge $\{2,3,4,5,6,7,8\}$ with $\{9\}$ and obtain $\{2,3,4,5,6,7,8,9\}$. Therefore, the next execution of line 6 will get $p = 9$. After the time interval $[0.92, 0.96)$ is scheduled and e_9 is updated to 0.96, so the algorithm will not do any union. The last execution finds $p = 9$ again, and a loop break is performed.

Now, we we analyze the time complexity of the algorithm.

Lemma 1 *Each set always contains continuous integers.*

Proof It can be proved by induction. At the beginning, each skeleton set is a continuous integer set. During the running of the algorithm, the union operation always merges two nearby continuous integer sets to form a larger continuous integer set.

Lemma 2 *There are at most $m - 2$ unions.*

Proof It is because there are only $m - 1$ sets.

Lemma 3 *There are at most $2(m - 2) + n$ finds.*

Proof Some $m - 2$ finds are from finding the set containing $p + 1$ during each union. Note that there is no need to perform a find operation to find the set containing p for union, because p is just the name of such a set, where the set contains continuous integers with p as the largest element. The other $(m-2)+n$ finds are from searching for earliest canonical time intervals beginning at or after time t_i. This can be analyzed in the following way: Let z_k be the number of times to search for an earliest non-empty canonical time interval when processing job j_k. Let w_k be the number of unions that are performed when processing job j_k. We have $z_k \leq w_k + 1$, because each of the first $z_k - 1$ finds must accompany a union. Therefore,

$$\sum_{1 \leq k \leq n} z_k \leq \sum_{1 \leq k \leq n} (w_k + 1) = \sum_{1 \leq k \leq n} w_k + n \leq (m - 2) + n.$$

Since these unions and finds are operated on the sets of integer intervals, such an interval union-find problem can be solved in $O(m + n)$ time in the unit-cost RAM model using Gabow and Tarjan's algorithm [12]. Note that $m = O(n)$, so the total time complexity is $O(n)$. Theorem 1 holds.

If the union-find algorithm is implemented in the pointer machine model [8] using the classical algorithm of Tarjan [25], the complexity of our s-schedule algorithm will become $O(n\alpha(n))$ where $\alpha(n)$ is the one-parameter inverse Ackermann function.

Note that, the number of finds can be further reduced with a more careful implementation of the algorithm as follows (but the asymptotic complexity will not change):

- Whenever the algorithm schedules a job j_k to run at a time interval $[e_p, b_k)$, the algorithm no longer needs to proceed to line 6 for the same job, because there will not be any idle time interval available before the deadline.
- For each job j_k, the first time to find a non-empty canonical time interval requires one find operation. In any of the later times to search for earliest non-empty canonical time intervals for the same job, there must be a union operation just performed. The p that determines the earliest non-empty canonical time interval $[e_p, t_{p+1})$ is just the name of that new set after that union, so a find operation is not necessary in this case. Note that the find operations that accompany the unions are still required.

Using the above implementation, the number of finds to search for earliest non-empty canonical time intervals can be reduced to n. Along with the $m - 2$ finds for unions, the total number of finds of this improved implementation is at most $(m - 2) + n$.

4 An $O(n^2)$ Continuous DVS Algorithm

We will first take a brief look at the previous best known DVS algorithm of Li, Yao and Yao [21]. As in [21], Define the "support" U of J to be the union of all job intervals in J. Define $\text{avr}(J)$, the "average rate" of J to be the total workload of J divided by $|U|$. According to Lemma 9 in [21], using $s = \text{avr}(J)$ to do an s-schedule will generate two nonempty subsets of jobs requiring speed at least s or less than s respectively in the optimal schedule unless the optimal speed for J is a constant s. The algorithm will recursively do the scheduling based on the two subsets of jobs. Therefore, at most n calls of s-schedules on a job set with at most n jobs are needed before we obtain the optimal schedule for the whole job set. The most time-consuming part of their algorithm is the s-schedules.

To apply our improved s-schedule algorithm for solving the continuous DVS scheduling problem, we need to make sure that the ranks of the deadlines and arrival times are known before each s-schedule call. It can be done in the following way: Before the first call, sort the deadlines and arrival times and obtain the ranks. In each of the subsequent calls, in order to get the new

ranks within the two subsets of jobs, a counting sort algorithm can be used to sort the old ranks in linear time. Therefore, the time to obtain the ranks is at most $O(n^2)$ for the whole algorithm. Based on the improved computation of s-schedules, the total time complexity of the DVS problem is now $O(n^2)$, improving the previous $O(n^2 \log n)$ algorithm of [21] by a factor of $O(\log n)$. We have the following theorem.

Theorem 2 *The continuous DVS scheduling problem can be solved in $O(n^2)$ time for n jobs in the unit-cost RAM model.*

5 Further Improvements

For the discrete DVS scheduling problem, we have an $O(n \log n)$ algorithm to calculate the optimal schedule by doing binary testing on the given d speed levels, improving upon the previously best known $O(dn \log n)$ [19]. To be specific, given the input job set with size n and a set of speeds $\{s_1, s_2, \ldots, s_d\}$, we first choose the speed $s_{d/2}$ to bi-partition the job set into two subsets. Then within each subset, we again choose the middle speed level to do the bi-partition. We recursively do the bi-partition until all the speed levels are handled. In the recursion tree thus built, we claim that the re-sorting for subproblems on the same level can be done in $O(n)$ time which implies that the total time needed is $O(n \log d + n \log n) = O(n \log \max\{d, n\})$. The claim can be shown in the following way. Based on the initial sorting, we can assign a new label to each job specifying which subgroup it belongs to when doing bi-partitioning. Then a linear scan can produce the sorted list for each subgroup.

6 Conclusion

In this paper, we improve the time for computing the optimal continuous DVS schedule from $O(n^2 \log n)$ to $O(n^2)$. The major improvement happens in the computation of s-schedules. Originally, the s-schedule computation is done in an online fashion where the execution time is allocated from the beginning to the end sequentially and the time assigned to a certain job can be gradually decided. While in this work, we allocate execution time to jobs in an offline fashion. When jobs are sorted by deadlines, job j_i's execution time is totally decided before we go on to consider j_{i+1}. Then by using a suitable data structure and conducting a careful analysis, the computation time for s-schedules improves from $O(n \log n)$ to $O(n)$. We also design an algorithm to improve the computation of the optimal schedule for the discrete model from $O(dn \log n)$ to $O(n \log \max\{d, n\})$.

References

1. S. Albers. Energy-Efficient Algorithms. *Communications of the ACM*, 53(1), pp. 86-96, 2010.

2. S. Albers. Algorithms for Dynamic Speed Scaling. *(STACS 2011)*, pp. 1-11.
3. S. Albers and A. Antoniadis: Race to idle: new algorithms for speed scaling with a sleep state. *(SODA 2012)*, pp. 1266-1285.
4. S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *FOCS'98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 534–544, 1998.
5. N. Bansal, D. P. Bunde, H. L. Chan, and K. Pruhs. Average Rate Speed Scaling. *In Proceedings of the 8th Latin American Theoretical Infomatics Symposium*, volume 4957 of LNCS, 2008, pp. 240-251.
6. N. Bansal, H. L. Chan, T. W. Lam and L.-K. Lee. Scheduling for speed bounded processors. *In Proceedings of the 35th International Symposium on Automata, Languages and Programming*, 2008, pp. 409-420.
7. N. Bansal, T. Kimbrel, and K. Pruhs. Dynamic Speed Scaling to Manage Energy and Temperature. *In Proceedings of the 45th Annual Symposium on Foundations of Computer Science*, 2004, pp. 520-529.
8. A. M. Ben-Amram. What is a "pointer machine"?. SIGACT News 26, 2 (June 1995), 88-95.
9. D. P. Bunde. Power-Aware Scheduling for Makespan and Flow. *In Proceedings of the 18th annual ACM symposium on Parallelism in algorithms and architectures*, 2006, pp. 190-196.
10. H. L. Chan, W. T. Chan, T. W. Lam, L. K. Lee, K. S. Mak, and P. W. H. Wong. Energy Efficient Online Deadline Scheduling. *In Proceedings of the 18th annual ACM-SIAM symposium on Discrete algorithms*, 2007, pp. 795-804.
11. W. T. Chan, T. W. Lam, K. S. Mak, and P. W. H. Wong. Online Deadline Scheduling with Bounded Energy Efficiency. *In Proceedings of the 4th Annual Conference on Theory and Applications of Models of Computation*, 2007, pp. 416-427.
12. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 246–251, New York, NY, USA, 1983. ACM.
13. I. Hong, G. Qu, M. Potkonjak and M. B. Srivastavas. Synthesis techniques for low-power hard real-time systems on variable voltage processors. *In Proceedings of the IEEE Real-Time Systems Symposium*, 1998, pp. 178-187.
14. S. Irani, R. K. Gupta and S. Shukla. Algorithms for Power Savings. *ACM Transactions on Algorithms*, 2007, 3(4).
15. S. Irani and K. Pruhs. Algorithmic Problems in Power Management. *ACM SIGACT News*, 2005, 36(2): pp. 63-76.
16. T. Ishihara and H. Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. *In Proceedings of International Symposium on Low Power Electronics and Design*, 1998, pp. 197-202.
17. W. Kwon and T. Kim. Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors. *In Proceedings of the 40th Conference on Design Automation*, 2003, pp. 125-130.
18. T. W. Lam, L. K. Lee, I. K. K. To, and P. W. H. Wong. Energy Efficient Deadline Scheduling in Two Processor Systems. *In Proceedings of the 18th International Symposium on Algorithm and Computation*, 2007, pp. 476-487.
19. M. Li and F. F. Yao. An Efficient Algorithm for Computing Optimal Discrete Voltage Schedules. *SIAM Journal on Computing*, 2005, 35(3): pp. 658-671.
20. M. Li, Becky J. Liu, F. F. Yao. Min-Energy Voltage Allocation for Tree-Structured Tasks. *Journal of Combinatorial Optimization*, 2006, 11(3): pp. 305-319.
21. M. Li, A. C. Yao, and F. F. Yao. Discrete and Continuous Min-Energy Schedules for Variable Voltage Processors. *In Proceedings of the National Academy of Sciences USA*, 2006, 103(11), pp. 3983-3987.
22. K. Pruhs and C. Stein. How to Schedule When You Have to Buy Your Energy. *in the Proceedings of the 13th International Workshop on Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, 2010, pp. 352-365.
23. K. Pruhs, P. Uthaisombut, and G. Woeginger. Getting the Best Response for Your Erg. *In Scandanavian Workshop on Algorithms and Theory*, 2004, pp. 14-25.

24. G. Quan and X. S. Hu. Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors. *Proceedings of the 38th Design Automation Conference*, 2001, pp. 828-833.
25. R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM 22, 2 (April 1975), 215-225.
26. W. Wu, M. Li and E. Chen. Min-Energy Scheduling for Aligned Jobs in Accelerate Model. *Theoretical Computer Science*, 2011, 412(12-14), pp. 1122-1139.
27. 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.
28. H. S. Yun and J. Kim. On Energy-Optimal Voltage Scheduling for Fixed-Priority Hard Real-Time Systems. *ACM Transactions on Embedded Computing Systems*, 2003, 2(3): pp. 393-430.