Min-Energy Voltage Allocation for Tree-Structured Tasks

Minming Li[1,⋆], Becky Jie Liu[2], and Frances F. Yao[2]

[1] Department of Computer Science, Tsinghua University
liminming98@mails.tsinghua.edu.cn
[2] Department of Computer Science, City University of Hong Kong
jliu@cs.cityu.edu.hk, csfyao@cityu.edu.hk

Abstract. We study job scheduling on processors capable of running at variable voltage/speed to minimize energy consumption. Each job in a problem instance is specified by its arrival time and deadline, together with required number of CPU cycles. It is known that the minimum energy schedule for n jobs can be computed in $O(n^3)$ time, assuming a convex energy function. We investigate more efficient algorithms for computing the optimal schedule when the job sets have certain special structures. When the time intervals are structured as trees, the minimum energy schedule is shown to have a succinct characterization and is computable in time $O(P)$ where P is the tree's total path length. We also study an on-line average-rate heuristics AVR and prove that its energy consumption achieves a small constant competitive ratio for nested job sets and for job sets with limited overlap. Some simulation results are also given.

1 Introduction

Portable electronic devices have in recent years seen a dramatic rise in availability and widespread use. This is partly brought on by new technologies enabling integration of multiple functions on a single chip (SOC). However, with increasing functionality also comes ever greater demand for battery power, and energy efficient implementations have become an important consideration for portable devices.

Generally speaking, the main approach is to trade execution speed for lower energy consumption while still meeting all deadlines. A number of techniques have been applied in embedded systems to reduce energy consumption. Modes such as idle, standby and sleep are already available in many processors. More energy savings can be achieved by applying Dynamic Voltage Scaling (DVS) techniques on variable voltage processors, such as the Intel *SpeedStep* technology [2] currently used in Intel's notebooks. With the newest *Foxon* technology

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

L. Wang (Ed.): COCOON 2005, LNCS 3595, pp. 283–296, 2005.
© Springer-Verlag Berlin Heidelberg 2005

(announced in February 2005), future Intel server chips can choose from as many as 64 speed grades, up from two or three in SpeedStep.

The associated scheduling problem for variable voltage processors has generated much interest, and an extensive literature now exists on this research topic. One of the earliest models for energy-efficient scheduling was introduced by Yao, Demers and Shenker in [1]. They described a minimum-energy off-line preemptive scheduling algorithm, with no restriction on the power consumption function except convexity. Also, two on-line heuristics AVR (Average Rate) and OPA (Optimal Available) were introduced, and it was shown that AVR has a competitive ratio of at most 8 for all job sets.

Under various related models and assumptions, more theoretical research has been done on minimum energy scheduling. For jobs with fixed priority, it was shown to be NP-hard to calculate the min-energy schedule and an FPTAS was given for the problem by Yun and Kim [6]. For discrete variable voltage processors, a polynomial time algorithm for finding the optimal schedule was given by Kwon and Kim [3]. Recently a tight competitive ratio of 4 was proven for the Optimal Available heuristic (OPA) [7]. Another related model which focuses on power down energy was considered in [9].

On the practical side, the problem has been considered under different systems constraints. For example, in [5] a task slowdown algorithm that minimizes energy consumption was proposed, taking into account resource standby energy as well as processor leakage. By considering limitations of current processors, such as transition overhead or discrete voltage levels, it was shown how to obtain a feasible (although non-optimal) schedule [4].

In this paper, we present efficient algorithms for computing the optimal schedules when the time intervals of the tasks have certain natural structures. These include tree-structured job sets which can arise from executing recursive procedure calls, and job sets with limited time overlap among tasks. We derive succinct characterizations of the minimum-energy schedules in these cases, leading to efficient algorithms for computing them. For general trees we obtain an $O(P)$ algorithm where P is the tree's total path length. In special cases when the tree is a nested chain or has bounded depth, the complexity reduces to $O(n)$. We also study the competitive ratio of on-line heuristic AVR for common-deadline job sets and limited-overlap job sets. A tight bound of 4 is proved in the former case, and an upper bound of 2.72 is proved in the latter case. Finally, we establish a lower bound of $\frac{17}{13}$ on the competitive ratio for any online schedule, assuming that time is discrete.

The significance of our work is twofold. First, the cases we consider, such as tree-structured tasks or common-deadline tasks, represent common job types. A thorough understanding of their min-energy schedules and effective heuristics can serve as useful tools for solving other voltage scheduling problems. Secondly, the characterization of these optimal solutions give rise to nice combinatorial problems of independent interest. For example, the optimal voltage scheduling for tree job sets can be viewed as a special kind of weight-balancing problem on trees (see Section 3).

The paper is organized as follows. We first review the scheduling model, off-line optimal schedule, and on-line AVR heuristic in Sections 2. In Section 3, we consider tree job sets and develop effective characterizations and algorithms for finding the optimal schedule. We also point out two special cases, the nested chain and the common deadline cases and give particularly compact algorithms for them. Analysis of competitive ratio is presented in Section 4 and lower bound of competitive ratio is discussed in Section 5. After presenting some simulation results in Section 6, we finish with concluding remarks and open problems in Section 7.

2 Preliminaries

We first review the minimum-energy scheduling model described in [1], as well as the off-line optimal scheduling algorithm and AVR online heuristic. For consistency, we adopt the same notions as used in [1].

2.1 Scheduling Model

Let J be a set of jobs to be executed during time interval $[t_0, t_1]$. Each job $j_k \in J$ is characterized by three parameters.

- a_k arrival time,
- b_k deadline, $(b_k > a_k)$
- R_k required number of CPU cycles.

A schedule S is a pair of functions $\{s(t), job(t)\}$ defined over $[t_0, t_1]$. Both $s(t)$ and $job(t)$ are piecewise constant with finitely many discontinuities.

- $s(t) \geq 0$ is the processor speed at time t,
- $job(t)$ defines the job being executed at time t (or idle if $s(t) = 0$).

A feasible schedule for an instance J is a schedule S that satisfies $\int_{a_k}^{b_k} s(t)\delta(job(t), j_k)dt = R_k$ for all $j_k \in J$ (where $\delta(x, y)$ is 1 if $x = y$ and 0 otherwise). In other words, S must give each job j the required number of cycles between its 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 $E(S) = \int_a^b P(s(t))dt$.

The goal of the scheduling problem is to find, for any given problem instance, a feasible schedule that minimizes $E(S)$. We remark that it is sufficient to focus on the computation of the optimal speed function $s(t)$; the related function $job(t)$ can be obtained with the earliest-deadline-first (EDF) principle.

2.2 The Minimum Energy Scheduler

We consider the off-line version of the scheduling problem and give a characterization of an energy-optimal schedule for any set of n jobs.

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

***Definition* 1** *Define the intensity of an interval $I = [z, z']$ to be $g(I) = \frac{\sum R_k}{z'-z}$ where the sum is taken over $J_I = \{$all jobs j_k with $[a_k, b_k] \subseteq [z, z']\}$.*

Clearly, $g(I)$ is a lower bound on the average processing speed, $\int_z^{z'} s(t)dt/(z'-z)$, that is required by any feasible schedule over the interval $[z, z']$. By convexity of the power function, any schedule using constant speed $g(I)$ over I is necessarily optimal on that interval. A critical interval I^* is an interval with maximum intensity $\max g(I)$ among all intervals I. It can be shown that the jobs in J_{I^*} allow a feasible schedule at speed $g(I^*)$ with the EDF principle. Based on this, Algorithm OS for finding the optimal schedule is given below and it can be implemented in $O(n^3)$ time [1].

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 the jobs in J_{I*} at speed s by EDF policy
  J ← J − J_{I*}
  for all j_k ∈ J do
    if b_k ∈ [z, z'] then
      b_k ← z
    else if b_k ≥ z' then
      b_k ← b_k − (z' − z)
    end if
    Reset arrival times similarly
  end for
until J is empty
```

2.3 On-Line Scheduling Heuristics

Associated with each job j_k are its *average-rate* $d_k = \frac{R_k}{b_k - a_k}$ and the corresponding step function

$$d_k(t) = \begin{cases} d_k \text{ if } t \in [a_k, b_k] \\ 0 \quad \text{elsewhere.} \end{cases}$$

Average Rate Heuristic computes the processor speed as the sum of all available jobs' average-rate. See Figure 1. At any time t, processor speed is set to be

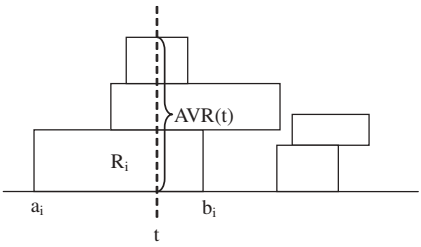

Fig. 1. Example of AVR heuristic

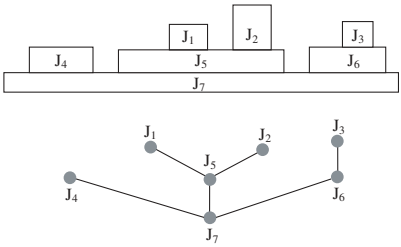

Fig. 2. Example of a tree job set

$s(t) = \sum_k d_k(t)$. Then, it uses the EDF policy to choose among available jobs for execution. Obviously this approach yields a feasible schedule.

Since the analysis of competitive ratio depends on the precise form of $P(s)$, our analysis is conducted under the assumption that $P(s) = s^2$. For a given job set J, let $OPT(J)$ denote the energy consumption of the optimal schedule, and let $AVR(J) = \int (\sum_k d_k(t))^2 dt$ denote the energy consumption of the AVR heuristic schedule. The competitive ratio of the heuristic is defined as the least upper bound of $AVR(J)/OPT(J)$ over all J.

It has been proved in [1] that AVR heuristic has a competitive ratio of at most 8 for any job set.

3 Optimal Voltage Schedule for Tree Job Sets

We consider the scheduling instance when the job intervals are properly nested as in a tree structure. The motivations are twofold: 1) such job sets arise naturally in practice, e.g. in the execution of recursively structured programs; 2) the characterization of the optimal speed function is nontrivial and leads to interesting combinatorial problems of independent interest.

3.1 Characterization of Optimal Schedule for Trees

***Definition* 2** *A job set J is called a tree job set if for any pair of job intervals I_j and I_k, one of the following relations holds: $I_j \cap I_k = \emptyset$, $I_j \subseteq I_k$ or $I_k \subseteq I_j$.*

For a tree job set, the inclusion relationship among job intervals can be represented graph-theoretically by a tree where each node corresponds to a job. See Figure 2. Job j_i is a descendant of job j_k iff $I_i \subseteq I_k$; if furthermore no other job $j_{i'}$ satisfies $I_i \subseteq I_{i'} \subseteq I_k$ then job j_i is a child of job j_k.

Interesting special cases of trees include jobs forming a single path (a nested chain), and jobs sharing a common deadline (or symmetrically, common arrival time). We will present linear algorithms for these cases in the next subsection, after first describing properties and algorithms for the general tree case. As remarked before, it is sufficient to focus on the computation of the optimal

speed function $s(t)$; the related function $job(t)$ can be obtained with the EDF principle.

We will prove a key lemma that is central to constructing optimal schedules for tree job sets. Suppose an optimal schedule has been given for a tree job set without its root node, we consider how to update the existing optimal schedule when a root node is added.

Consider any job set J consisting of n jobs (not necessary tree-structured), and an additional new job j_{n+1} with the property that $[a_{n+1}, b_{n+1}] \supseteq [a_k, b_k]$ for any $j_k \in J$. We will show that the optimal schedule $s'(t)$ for job set $J' = J \cup \{j_{n+1}\}$ is uniquely determined from the optimal schedule $s(t)$ of J and description of j_{n+1}. That is, information such as $[a_k, b_k]$ and R_k for $j_k \in J$ is not needed for computing $s'(t)$ from $s(t)$. This property will enable us to construct the optimal schedule for a tree job set efficiently in a bottom-up procedure.

To prove the above claim, we compare the selection of critical intervals for $s'(t)$ versus that for $s(t)$. Suppose $s(t)$ consists of m critical intervals $I_1^*, \ldots, I_m^*$ with lengths $l_1, \ldots, l_m$ and speeds $s_1 > \cdots > s_m$. Comparing the computation of $s'(t)$ by Algorithm OS versus that of $s(t)$, we note that the only new candidate for critical intervals in each round is $I_{n+1} = [a_{n+1}, b_{n+1}]$. (Due to the fact $[a_{n+1}, b_{n+1}] \supseteq [a_k, b_k]$, no other intervals of the form $[a_{n+1}, b_k]$ or $[a_k, b_{n+1}]$ is a feasible candidate.) Moreover, as soon as I_{n+1} is selected as the critical interval, all currently remaining jobs will be executed (since their intervals are contained in I_{n+1}) and Algorithm OS will terminate.

By examining the intensity $g(I_{n+1})$ of interval I_{n+1} in each round i and comparing it with the speed s_i, we can determine exactly in which round I_{n+1} will be selected as the critical interval. This index i will be referred to as the *terminal index* (of job j_{n+1} relative to $s(t)$). To find the terminal index i, it is convenient to start with the maximum $i = m + 1$ and search backwards. Let $g_i(I_{n+1})$ denote the intensity of I_{n+1} as would be calculated in the i-th round of Algorithm OS. Using $w_k = s_k l_k$ to denote the workload executed in the k^{th} critical interval of $s(t)$ for $k = 1, \ldots, m$, and letting $w_{m+1} = R_{n+1}$ and $l_{m+1} = |I_{n+1}| - \sum_{k=1}^{m} l_k$, we can write $g_i(I_{n+1})$ as $g_i(I_{n+1}) = (\sum_{k=i}^{m+1} w_k)/(\sum_{k=i}^{m+1} l_k)$.

It follows that the terminal index is the largest i, $1 \le i \le m+1$, for which the following holds: $g_i(I_{n+1}) \ge s_i$ and $g_{i-1}(I_{n+1}) < s_{i-1}$ (where we set $s_0 = \infty$, $g_0(I_{n+1}) = 0$, and $s_{m+1} = 0$ as boundary conditions).

We have proven the following main lemma for tree job sets.

Lemma 1 *Let the optimal schedule $s(t)$ for a job set J be given, consisting of speeds $s_1 > \cdots > s_m$ over intervals $I_1^*, \ldots, I_m^*$. If a new job j_{n+1} satisfies $[a_{n+1}, b_{n+1}] \supseteq [a_k, b_k]$ for all $j_k \in J$, then the optimal schedule $s'(t)$ for $J \cup \{j_{n+1}\}$ consists of speeds $s'_1 > s'_2 > \cdots > s'_i$ where*
1) i is the terminal index of j_{n+1} relative to $s(t)$,
2) critical intervals from 1 up to $i-1$ are identical in $s(t)$ and $s'(t)$, and
3) the i-th critical interval for $s'(t)$ has speed $s'_i = g_i(I_{n+1})$ over $I_{n+1} - \bigcup_{k=1}^{i-1} I_k^*$.

A procedure corresponding to the above lemma is given in Algorithm *Merge*, which updates an optimal schedule when a root node is added. The exact method

for finding the terminal index will be discussed in the next subsection. Based on *Merge*, we obtain a recursive algorithm for finding the optimal schedule for a tree job set as given in Algorithm *OST*. We also observe the following global property of a tree-induced schedule.

Lemma 2 *In the optimal schedule for a tree job set, the execution speeds of jobs along any root-leaf path form a non-decreasing sequence.*

Proof. By Lemma 1, for a tree job set J' consisting of a node j_{n+1} and all of its descendants, the execution speed of j_{n+1} is the minimum among J' since it defines the last critical interval. The lemma holds by treating every node as the root of some subtree.

3.2 Finding Terminal Indices for Trees

Algorithm OST gives a recursive procedure for constructing optimal schedule for a tree job set. The important step is to carry out $Merge(S_{J-\{r\}}, r)$ at every node r of the tree by finding the correct terminal index i. Naively, it would seem that sorting the execution speeds of all of $r's$ descendants is necessary for finding the terminal index quickly. It turns out that sorting (an $O(n \log n)$ process) can be replaced by median-finding (an $O(n)$ process [8]) as we show next. This enables us to achieve $O(P)$ complexity for the overall algorithm where P is the tree's total path length. For trees of bounded depth this gives an $O(n)$ algorithm. In the following discussion, we denote the optimal schedule of job set J as S_J.

Algorithm 2 *Merge*

Input: Optimal schedule S for J, new job j_{n+1}
Output: Optimal schedule S' for $J' = J \cup \{j_{n+1}\}$

$S' \leftarrow S$
$i \leftarrow Find(S, j_{n+1})$ {find terminal index i}
In S', replace $s_i, s_{i+1}, \ldots, s_m$ with $s'_i = g_i(I_{n+1})$
Return S'

Algorithm 3 *OST (Optimal Scheduling for Tree)*

Input: root r of tree job set J
Output: Optimal Schedule S_J for J

initialize $S_{J-\{r\}}$ to be $\emptyset$
for all Children ch_k of r **do**
  $S_{J-\{r\}} \leftarrow S_{J-\{r\}} \cup OST(ch_k)$
end for
$S_J \leftarrow Merge(S_{J-\{r\}}, r)$
Return S

Theorem 1 *Algorithm OST can compute an optimal schedule for any tree job set in $O(P)$ time where P is the total path length of the tree.*

Proof. For every $Merge$ operation, we can find the terminal index by performing a binary search on the median speed in S_{J-r}. That is, we find the median speed s_k and then decide in which half to search further. See Algorithm 4. Finding the median of a list of t items costs $O(t)$ time. Calculation of the associated $g(I_{n+1})$ value is also $O(t)$. The total cost of a binary search for the terminal index thus amounts to a geometric series whose sum is bounded by $O(t)$. Therefore, the cost of $Merge(S_{J-\{r\}}, r)$ is proportional to the number of descendant nodes of r. Hence the total cost of $Merge$ over all nodes of the tree is upper bounded by the tree's total path length.

Algorithm 4 *Find (by Median Search)*

Input: Schedule S consisting of speed $\{s_1, s_2, \cdots, s_m\}$ in unsorted manner, new job j_{n+1}

Output: Index i such that $g_i(I_{n+1}) \geq s_i$ and $g_{i-1}(I_{n+1}) < s_{i-1}$.

```
s_0 ← ∞
s_{m+1} ← 0
Find median value s_k in S
while k isn't the terminal index do
  if g_k(I_{n+1}) < s_k then
    S ← {s_j | j > k, s_j ∈ S}
  end if
  if g_{k-1}(I_{n+1}) ≥ s_{k-1} then
    S ← {s_j | j < k, s_j ∈ S}
  end if
  Find median speed s_k in S
end while
Return k
```

3.3 Finding Terminal Indices for Chains

For trees of depth $O(n)$, the above algorithm can have worst case complexity $O(n^2)$. However, we will show that for a nested chain of n jobs (corresponding to a single path of depth n), its optimal schedule can still be computed in $O(n)$ time. Here, instead of using repeated median-finding, Algorithm *Merge* will keep the speeds sorted and use a linear search to find the terminal index. We note that, without loss of generality, the n nested jobs can be first shifted so they all have a common deadline. (This is because the intersection relationship among time intervals have not been altered.) See Figure 3. Thus it is sufficient to describe an $O(n)$ algorithm for job sets with a common deadline.

Theorem 2 *The optimal schedule for a job set corresponding to a nested chain can be computed in $O(n)$ time.*

Proof. Let the job intervals be $[a_{n+1}, b] \supseteq [a_n, b] \supseteq \cdots \supseteq [a_1, b]$. We implement $Find(S_{J-\{j_{n+1}\}}, j_{n+1})$ with a linear search for the terminal index, starting with

the lowest speed in $S_{J-\{j_{n+1}\}}$. The key observation is that, as can be proven by induction, the speed function $s(t)$ is piecewise increasing from left to right. Hence the search for the terminal index can proceed from left to right, computing $g_k(I_{n+1})$ one by one from the smallest s_k. Notice that computing $g_k(I_{n+1})$ with knowledge of $g_{k-1}(I_{n+1})$ only costs constant operations. Furthermore, if we needed to compute g for u consecutive $k's$ before arriving at the right terminal index, then the total number of critical intervals will also have decreased by $u-1$. Suppose the Merge operation for the j-th job in the chain (starting from the leaf) computes g for u_j times, we have $\sum_{k=1}^{n}(u_k-1) \leq n$, which implies that $\sum_{k=1}^{n} u_k \leq 2n$. Therefore, the algorithm can finish computing the optimal schedule in $O(n)$ time.

The linear search procedure described above is given in Algorithm 5, and one execution of this *Find* procedure is illustrated in Figure 4.

Algorithm 5 *Find (by Linear Search)*

Input: Schedule S = $\{s_1 > s_2 > \cdots > s_m\}$, new job j_{n+1}
Output: Index i such that $g_i(I_{n+1}) \geq s_i$ and $g_{i-1}(I_{n+1}) < s_{i-1}$.

```
s_0 ← ∞
s_{m+1} ← 0
for k = m downto 1 do
  if g_k(I_{n+1}) < s_k then
    Return k + 1
  end if
end for
```

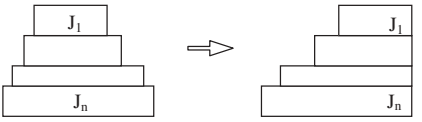

Fig. 3. Transforming chain job set into common deadline job set

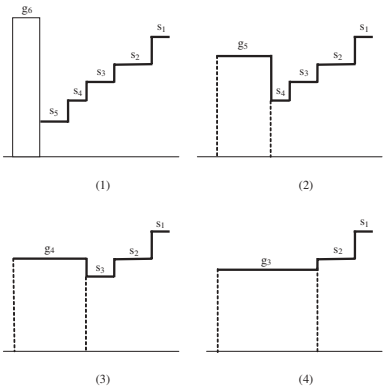

Fig. 4. Scheduling of a set of jobs with common deadline

3.4 A Weight Balancing Problem for Trees

We have presented two different strategies for implementing the *Find* operation on trees, as a subroutine used for computing the optimal voltage schedule. We

can formulate the problem as a pure weight balancing problem for trees, whose solution will provide alternative algorithms for computing the optimal schedule.

We start with a tree where each node is associated with a pair of weights (w_k, l_k). The goal is to adjust the weights so that the ratio $s_k = w_k/l_k$ along any root-leaf path will be monotonically non-decreasing (see Lemma 2). The rule for modifying the weights is to proceed recursively and, at each node r, 'merge' $r's$ weights with those of its descendants with smallest s_k so that their new 'average' ratio, as defined by $(\sum w_k)/\sum l_k)$, will satisfy the monotonicity condition. The challenge is to find a suitable data structure that supports the selection of $r's$ descendants for the weight balancing . Two different solutions to this problem were considered in Theorem 1 and 2 respectively. Are there other efficient methods?

4 Analysis of Competitive Ratio

We will analyze the performance of AVR versus OPT for several types of job sets. For convenience of reference, we state the definition of these job sets in the following.

***Definition* 3** *A Job set J is called*

i) a chain job set if $a_1 \leq a_2 \leq ... \leq a_n$ and $b_1 \geq b_2 \geq ... \geq b_n$

ii) a common deadline job set if $a_1 \leq a_2 \leq ... \leq a_n$ and $b_1 = b_2 = ... = b_n$

iii) a two-overlap job set if $I_i \cap I_{i+2} = \emptyset$ and $a_1 \leq a_2 \leq ... \leq a_n$ and $b_1 \leq b_2 \leq ... \leq b_n$.

4.1 Chain Job Set

Theorem 3 *For any nested chain job set J, $AVR(J) \leq 4\ OPT(J)$.*

This bound of 4 is tight, as an example J provided in [1] actually achieves $AVR(J) = 4\ OPT(J)$. In this example, the ith job has interval $[0, 1/n]$ and density $d_i = (n/i)^{3/2}$, this job set has competitive ratio 4 when $n \to \infty$.

It is obvious that transforming a chain job set into a common deadline job set by shifting preserves both $AVR(J)$ and $OPT(J)$, hence does not affect the competitive ratio. See Fig. 3. Thus we only need to focus on the competitive ratio for the common deadline case. Given a common deadline job set J, the algorithm in Theorem 2 will produce an optimal schedule with exactly one execution interval for each job $j_i \in J$. Denote the execution interval by $[c_i, c_{i+1}]$ where $c_1 = a_1$, $c_i \geq a_i$ and $c_{n+1} = b$. Given J, define J' to be the same as J except $a'_i = c_i$ for all i. We call J' the *normalized* job set for J.

We make use of the following algebraic relation; the proof is omitted here.

Lemma 3 *Let X and X' be two positive constants such that $\int_a^b X = \int_{a'}^b X'$ where $a \leq a' \leq b$. If $Y(t)$ is a monotone function such that $Y(t) \leq Y(t')$ for $t \leq t'$, then $\int_a^b (X + Y(t))^2 \leq \int_{a'}^b (X' + Y(t))^2 + \int_a^{a'} Y(t)^2$.*

Lemma 4 *Given a common deadline job set J, let J' be the normalized job set for J. Then we have*
1) $OPT(J) = OPT(J')$
2) $AVR(J) \leq AVR(J')$.

Proof. Property 1) is straightforward by the definition of J'. Property 2) can be proved by applying lemma 3 to the jobs inductively.

Proof of Theorem 3. We first convert J into a common deadline job set. Let J' be the normalized job set for J. By lemma 4, $AVR(J) \leq AVR(J')$ and $OPT(J) = OPT(J')$. According to Theorem 2 in [1], this will result in a competitive ratio of at most 4 for J'. Combining with $AVR(J) \leq AVR(J')$, we obtain $AVR(J) \leq 4\ OPT(J)$.

4.2 Two-Overlap Job Set

We first consider the simple case of a two-job instance and show that $AVR(J) \leq 1.36\ OPT(J)$ (proof omitted here). It is then used as the basis for the n-job case.

Lemma 5 *For any job set consisting of two jobs, $AVR(J) \leq 1.36\ OPT(J)$.*

Theorem 4 *For any two-overlap job set, $AVR(J) \leq 2.72\ OPT(J)$.*

Proof. Denote the two-job instance $\{j_i, j_{i+1}\}$ as J_i for $0 \leq i \leq n$. (We also introduce two empty jobs j_0 and j_{n+1}.) We have $AVR(J) = \sum_{i=0}^{n} AVR(J_i) - \sum_{i=1}^{n} d_i^2 t_i$. On the other hand, using the result for two-job sets, we have $\sum_{i=0}^{n} AVR(J_i) \leq 1.36 \sum_{i=0}^{n} OPT(J_i)$. We observe that any two consecutive jobs in a two-overlap job set must use more energy in the optimal schedule than when they are scheduled alone. Thus, $\sum_{i=0}^{n} OPT(J_i) \leq 2\ OPT(J)$. Combining the above three relations, we obtain $AVR(J) \leq 2.72\ OPT(J) - \sum_{i=1}^{n} d_i^2 t_i$ which proves the theorem.

5 Lower Bound for Online Schedules

To prove a lower bound on the competitive ratio of all online schedules, we make the assumption that the processor time comes in discrete units, i.e. the processor must maintain the same speed over each time unit. Given any online scheduler, we will construct a two-job instance for which the scheduler's performance is no better than $\frac{17}{13}$ times optimal.

The first job arrives at time 0 and its interval lasts for two time units. Its requirement is two CPU cycles. Suppose the online schedule allocates two CPU cycles to the first job on the first time unit. We then just let the second job be an empty job, and the schedule's cost is already 2 times optimal.

Suppose the online schedule allocates one CPU cycle to the first job on the first time unit. We construct the second job as follows: it starts from the second time unit and lasts for one time unit and requires 3 CPU cycles. In this case the online schedule's cost is $\frac{17}{13}$ times the optimal. This proves that $\frac{17}{13}$ is a lower bound on the competitive ratio of all online heuristics.

6 Simulation Results

We have simulated the performance of AVR online heuristic in three different types of job sets: general, two-overlap and common deadline job sets. The following data are collected from 1000 randomly generated job sets of each type. Each job set consists of 100 random jobs: the arrival times and deadlines are uniformly distributed over a continuous time interval $[0, 100]$ in the general case, and suitably constrained in the other two cases. The required CPU cycles of each job is chosen randomly between 0 and 200.

Average, maximum and minimum competitive ratios for each of the three cases are shown in Table 5. For the general case, the maximum competitive ratio we observed is 1.47, which is far better than the theoretical bound of 8. The minimum observed ratio is always close to 1. Best among the three, the two-overlap case is where AVR excels, achieving average ratio of 1.16 and maximum ratio of 1.21. For the common deadline case, the maximum ratio encountered of 2.25 is also much better than the bound of 4 proved in Theorem 3.

Type of Job Set	Average	Maximum	Minimum
General	1.215	1.469	1.007
Two-overlap	1.160	1.206	1.088
Common Deadline	1.894	1.255	1.113

Fig. 5. Summary of competitive ratios from simulations

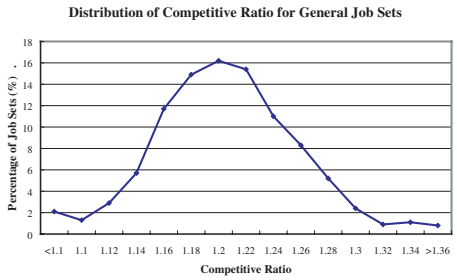

Fig. 6. Simulation result of competitive ratio for general job sets

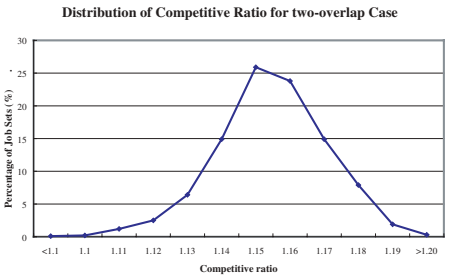

Fig. 7. Simulation result of competitive ratio for two-overlap job sets

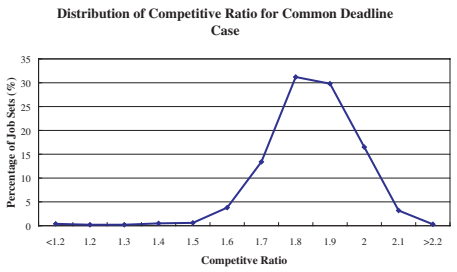

Fig. 8. Simulation result of competitive ratio for common-deadline job sets

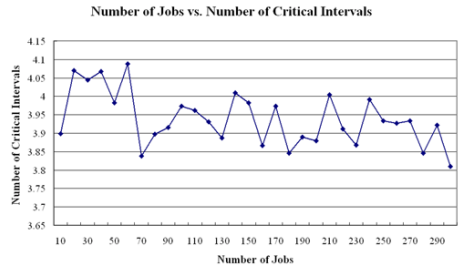

Fig. 9. Number of jobs vs. number of critical intervals

The detailed distributions of the competitive ratio obtained from the simulations are give in Figure 6, 7 and 8. The data suggest that the distributions are close to normal for all three types of job sets, with standard deviations of 0.0528, 0.0162 and 0.1336 respectively.

In the second simulation, we look for the growth of the number of critical intervals with respect to the number of jobs in the general case. For each n between 10 and 300, we randomly generate a set of n jobs, and then compute the average number of critical intervals over 1000 such job sets for each n. Quite surprisingly, this average number does not seem to grow noticeably with the number of jobs. According to Figure 9, the average number of critical intervals always lies within the range from 3.8 to 4.1 for any n between 10 to 300, with lowest value 3.81 for $n = 300$ and highest value 4.09 for $n = 60$.

7 Conclusion

In this paper, we considered the problem of minimum-energy scheduling for preemptive job sets, under the assumption that energy consumption is a convex function of processor speed. We first focus on the off-line scheduling of tree-structured job sets, where jobs are either properly nested or disjoint. Based on our observation that the optimal execution speeds form a non-decreasing sequence along any root-leaf path in the tree, we derived efficient bottom-up algorithm that computes the optimal voltage schedule for general tree-structured job sets. In addition, we gave an $O(n)$ algorithm for common-deadline or chain job sets.

We also studied the competitive ratio of on-line heuristic AVR for common-deadline job sets and limited-overlap job sets. A tight bound of 4 is proved in the former case, and an upper bound of 2.72 is proved in the latter case. Finally, we established that $\frac{17}{13}$ is a lower bound on the competitive ratio for any online schedule assuming that time is discrete.

Some interesting open problems remain. Our simulation results suggest that the number of critical intervals grows slowly with n; what exactly is the asymptotic rate? Can our findings for tree-structured job sets be generalized to other classes of job sets? Can the tree case itself be solved even more efficiently?

References

1. F. Yao, A. Demers and S. Shenker, *A Scheduling Model for Reduced CPU Energy*, Proceedings of the 36^{th} Annual Symposium on Foundations of Computer Science, 374-382, 1995.
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. W. Kwon and T. Kim, *Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors*, 40^{th} Design Automation Conference, 2003.
4. B. Mochocki, X. S. Hu and G. Quan, *A Realistic Variable Voltage Scheduling Model for Real-Time Applications*, IEEE/ACM International Conference on Computer-Aided Design, 2002.

5. R. Jejurikar and R. K. Gupta, *Dynamic Voltage Scaling for Systemwide Energy Minimization in Real-Time Embedded Systems*, International Symposium on Low Power Electronics and Design, 2004.
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(3): 393-430, 2003.
7. N. Bansal, T. Kimbrel and K. Pruhs, *Dynamic Speed Scaling to Manage Energy and Temperature*, Proceedings of the 45^{th} Annual Symposium on Foundations of Computer Science, 520-529, 2004.
8. M. Blum, R. Floyd, V. Pratt, R. Rivest and R. Tarjan, *Time Bounds for Selection*, Journal of Computer and System Sciences, 7:488-461, 1973.
9. J. Augustine, S. Irani and C. Swamy, *Optimal Power-Down Strategies*, Proceedings of the 45^th Annual Symposium on Foundations of Computer Science, 530-539, 2004.