Scheduling to Minimize Gaps and Power Consumption*

Erik D. Demaine† Mohammad Ghodsi‡§ MohammadTaghi Hajiaghayi¶
Amin S. Sayedi-Roshkhar‖ Morteza Zadimoghaddam†

Abstract

This paper considers scheduling tasks while minimizing the power consumption of one or more processors, each of which can go to sleep at a fixed cost α. There are two natural versions of this problem, both considered extensively in recent work: minimize the total power consumption (including computation time), or minimize the number of "gaps" in execution. For both versions in a multiprocessor system, we develop a polynomial-time algorithm based on sophisticated dynamic programming. In a generalization of the power-saving problem, where each task can execute in any of a specified set of time intervals, we develop a $(1 + \frac{2}{3}\alpha)$-approximation, and show that dependence on α is necessary. In contrast, the analogous multi-interval gap scheduling problem is set-cover hard (and thus not $o(\lg n)$-approximable), even in the special cases of just two intervals per job or just three unit intervals per job. We also prove several other hardness-of-approximation results. Finally, we give an $O(\sqrt{n})$-approximation for maximizing throughput given a hard upper bound on the number of gaps.

1 Introduction

Power is a growing concern in computer science, motivated by batteries increasing in capacity much more slowly than computation power, and by small mobile devices such as cell phones, PDAs, and sensors increasing in prevalence; see, e.g., the recent surveys in theory [IP05] and in practice [BBM00]. A common approach in practice for reducing power consumption is to add a "sleep state" which requires essentially no power but disallows computation. However, each transition from the sleep state to the regular, active state consumes a fixed amount of power, so we cannot simply go to sleep whenever computation is not required. The scheduling problem is thus to determine when to switch to the sleep state in order to minimize the total power consumption.

We consider two precise formulations of this problem. In *power minimization* [ISG03, AIS04], the objective is to minimize the total transition costs plus the total time spent in the active state. In *gap scheduling* [Bap06, FHKN06], the device goes into a sleep state whenever it is idle, and the

*A preliminary version of this paper appeared in *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, 2007, pages 46–54.

†MIT Computer Science and Artificial Intelligence Laboratory, edemaine@mit.edu, zadimoghaddam@gmail.com.

‡Department of Computer Engineering, Sharif University of Technology, Azadi St., Tehran, Iran, ghodsi@sharif.edu.

§Supported in part by Institute for Theoretical Physics and Mathematics (IPM) under grant numbers CS1385-2-01 and CS1384-6-01.

¶Computer Science Department, University of Maryland, College Park, MD 20742, USA, hajiagha@cs.umd.edu. Work done while at MIT.

‖Tepper School of Business, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA, ssayedir@cmu.edu

objective is to minimize the total number of transitions (because the time spent in the active state is fixed by the input jobs). Though minimizing total power (as in power minimization) is the most natural measure, minimizing the number of transitions (as in gap scheduling) seems stronger from the point of view of approximation algorithms.

In all previous work on these problems, tasks have arrival times, deadlines, and processing times, and the goal is to find a pre-emptive schedule that satisfies all deadlines. We consider a generalization, called *multi-interval scheduling*, in which each task has a list of one or more time intervals during which it can execute (e.g., when the necessary resources are available), and the goal is to complete every task. One practical special case of this generalization, also considered in this paper, is *multiprocessor scheduling* where each task can run on any processor, and individual processors can go into the sleep state. (To see that multiprocessor scheduling is a special case of interval scheduling, view the processor executions as laid out one after the other, so that idle gaps correspond; then a task with an arrival time and deadline becomes executable in an arithmetic sequence of time intervals.) This problem is particularly interesting given the increasing prevalence of multicore and multiprocessor architectures.

Previous work. It remained a challenging open problem for several years whether the offline version of both problems, in the one-interval case where each job has one interval (arrival time and deadline), was NP-hard or polynomially solvable. For exact solutions, offline power saving and offline gap scheduling are the same problem (they differ only with respect to approximation). This open problem was posed at the 2002 Dagstuhl Seminar on Online Algorithms, and in [ISG03, IP05].

Most preliminary work on this problem considered power saving, which is easier in terms of approximation. For offline power saving, Irani, Shukla, and Gupta [ISG03] developed a polynomial-time 3-approximation. For online power saving, where jobs become known only at their arrival time, Augustine, Irani, and Swamy [AIS04] obtained a $(3 + 2\sqrt{2})$-competitive strategy, while the best lower bound on competitive ratio is 2 [ISG03, Bap06].

The open problem was finally solved by Baptiste [Bap06], who gave a surprising and clever dynamic program that solves the offline one-interval problem (both power saving and gap scheduling) optimally in polynomial time. At his SODA 2006 talk, he posed a harder version of this open problem: what about multi-interval scheduling? This is the topic of our paper.

Offline one-interval gap scheduling also has a simple greedy 3-approximation algorithm [FHKN06]. The algorithm tries all possible gaps and chooses the largest gap that still leaves a feasible schedule (whose existence can be checked by maximum-cardinality matching). Then it removes this interval of time and repeats the process until no more gaps can be introduced. An initial analysis shows that this greedy algorithm is an $O(\lg n)$-approximation, by analogy to set cover, but with substantially more work, it is proved to be a 3-approximation.

Note that, without a power or gap objective, the offline one-interval problem is one of the most basic and fundamental scheduling problems. This problem has a simple optimal greedy solution, earliest deadline first, as well as other approaches via linear programming or bipartite matching.

Our focus is on offline problems, because multi-interval power saving and even one-interval gap scheduling are not interesting in the online context. If an online algorithm is guaranteed to find a feasible schedule whenever possible, then it must follow the earliest-deadline-first schedule, executing jobs whenever they become available. For multi-interval scheduling, the earliest-deadline-first schedule is not well-defined, and indeed any online algorithm cannot find a feasible schedule when possible. To see this, suppose two jobs come at time 0, one with intervals $[0, 1]$ and $[1, 2]$ and the other with intervals $[0, 1]$ and $[2, 3]$; we cannot tell which job to run at time 0, for fear of a third job coming at time 1 or 2 and requiring immediate execution. On the other hand, for one-interval gap scheduling, suppose n jobs arrive at time 0 with a deadline of $3n$, and n jobs arrive at times $n, n+2, n+4, \ldots$ with deadlines of one unit after their arrival. The optimal gap schedule waits for the n latter jobs to arrive and schedules the former n jobs in between the gaps, for a gap cost of $O(1)$. Any online solution guaranteed to find a feasible solution must schedule the former n jobs immediately, for a gap cost of n, to handle the instance where the latter jobs are instead $2n$ jobs arriving at times $n, n+1, n+2, \ldots$. So any correct online algorithm must have a competitive ratio of at least n.

Our results. Our first positive result (Section 2) is for the important case of (one-interval) multiprocessor scheduling, which can be viewed as a special case of multi-interval scheduling where each task can execute in an arithmetic sequence of time intervals with fixed period. Here we obtain a polynomial-time algorithm, nontrivially building on the dynamic program of [Bap06] by obtaining additional structure. Somewhat surprisingly, the running time of the dynamic is polynomial in both n and the number p of processors, not e.g. $n^{O(p)}$. Because our algorithm is exact, it solves both the power-saving and gap versions of the problem.

For multi-interval power saving with a transition cost of α, we develop a $(1+\frac{2}{3}\alpha)$-approximation algorithm (Section 3). This result builds on some classic previous work on set packing by Hurkens and Schrijver [HS89]. (In *maximum set packing*, given a collection of sets, the goal is to find a maximum subcollection of pairwise-disjoint sets.) We also show that no polynomial-time approximation factor can be independent of α unless P = NP (Section 4.2). Furthermore, if α is part of the input, we show that multi-interval power saving is set-cover hard (Section 4.1), and therefore cannot be approximated within a $o(\lg n)$ factor unless P = NP [Fei98].

Next we show through a series of reductions that multi-interval gap scheduling is set-cover hard, even in two special cases. In the first case (Section 5.1), each job has only two intervals (*two-interval gap scheduling*). In the second case (Section 5.2), each job has at most three intervals each of which is one unit in length (*three-unit gap scheduling*). Slightly more positive (Section 5.3), if each job has at most two intervals each of which is one unit in length (*two-unit gap scheduling*), then we show that the problem is equivalent to the variation where all intervals of all jobs are disjoint (*disjoint-interval gap scheduling*), which we prove cannot be approximated within any constant factor.

Finally, we consider swapping the roles of the hard constraint and the objective function in gap scheduling (Section 6). Namely, if we allow at most k gaps (i.e., at most k "restarts"), what is the maximum number of jobs that we can schedule? We give an $O(\sqrt{n})$-approximation algorithm for this *minimum-restart problem*. A simple example of this scenario is hiring a consultant who bills by the day. The consultant goes home whenever there is no work to do; if you ask the consultant back later that day, it counts as a new day (restarting). Each job can be executed at specified times during specified days, but not at night, so the consultant goes home each night. What can you get the consultant do given a hiring budget of only k days?

2 Multiprocessor Gap Scheduling: Homogeneous Arithmetic Intervals

In multiprocessor gap scheduling, we have p processors $P_1, P_2, \ldots, P_p$ and n jobs $j_1, j_2, \ldots, j_n$; each job j_i has an integer arrival time a_i, integer deadline d_i, and unit processing time. A *feasible schedule* assigns each job j_i to a unique processor/time pair (p_i, t_i) where the integer t_i satisfies $a_i \le t_i \le d_i$. A *gap* on processor P_q is a finite maximal interval of time during which no jobs are scheduled on P_q. Our goal is to determine whether there is a feasible schedule, and if so, to find one that minimizes the total number of gaps summed over all processors.

Building on Baptiste's clever dynamic program [Bap06], we obtain the following result:

Theorem 1 *There is a $(n^7 p^5)$-time algorithm for p-processor gap scheduling of n jobs.*

The p-processor problem can be seen as a special case of the multi-interval problem, where each job has p intervals and the intervals for a job are of the form $I, I+x, I+2x, \ldots, I+(p-1)x$. To see this connection, view the processor executions as laid out one after the other on the timeline, where each processor runs for less than x units.

Thus, as a consequence of Theorem 1, we also obtain a polynomial-time algorithm for arithmetic p-interval scheduling when the arithmetic series all have the same long period x. This result is in surprising contrast to our negative results: for example, 2-unit gap scheduling, where each job's intervals are necessarily arithmetic, is inapproximable within any constant factor. The only difference in this case is that the arithmetic series have different (and possibly small) periods.

Before we can describe our dynamic program, we need a structural result characterizing some optimal solutions:

Lemma 1 *Any feasible instance of multiprocessor gap scheduling has an optimal solution in which, if a job j is scheduled at time t on processor P_q, then all lower-numbered processors $P_1, P_2, \ldots, P_{q-1}$ are also occupied at time t.*

Proof: Permuting jobs and idleness among processors does not affect feasibility; it could only affect the number of gaps. Instead of counting gaps, we can count the jobs for which the previous time unit of their own processor lacks a job. For any time t, suppose that the optimal solution has ℓ jobs scheduled during time unit t and ℓ' jobs scheduled during time unit $t+1$. We know that at least $\ell' - \ell$ gaps must created at the boundary between times t and $t+1$. If we move the jobs at time t to the ℓ lowest-numbered processors $P_1, P_2, \ldots, P_\ell$, and move the jobs at time $t+1$ to the ℓ' lowest-numbered processors $P_1, P_2, \ldots, P_{\ell'}$, then the number of gaps in the schedule on the boundary between times t and $t+1$ is not more than $\ell' - \ell$. Therefore, by moving all jobs in this way, the total number of gaps does not increase. $\square$

Now we can present our dynamic program for multiprocessor gap scheduling:

Proof of Theorem 1: We use dynamic programming to solve the problem. Define $C_{t_1,t_2,k,q,\ell_1,\ell_2}$ to be the number of gaps in the optimal solution for our subproblem defined as follows. We want to schedule jobs $j_1, j_2, \ldots, j_k$ in the interval $I = [t_1, t_2]$. We are promised that the release time of each j_i is within I and that, among all jobs with release time in I, $j_1, j_2, \ldots, j_k$ are those with earliest deadlines. We are given that, at time t_2, we can only use processors $P_{q+1}, P_{q+2}, \ldots, P_p$ numbered greater than q: processors $P_1, P_2, \ldots, P_q$ are assumed to be occupied. We are required to schedule exactly ℓ_1 jobs at time t_1 and ℓ_2 jobs at time t_2. By Lemma 1, the former ℓ_1 jobs must be schedule on processors $P_1, P_2, \ldots, P_{\ell_1}$, while the latter ℓ_2 jobs must be scheduled on processors $P_{q+1}, P_{q+2}, \ldots, P_{q+\ell_2}$.

To solve the subproblem, we consider the job j_k that has the latest deadline among the jobs $j_1, j_2, \ldots, j_k$. For this job to actually be j_k, we presort the jobs by increasing deadline at the beginning of the algorithm. Suppose that j_k is scheduled at time t' and suppose that t' is maximal among all optimal solutions. Suppose that there are i jobs scheduled after t' and there are $k-i-1$ jobs (excluding j_k) scheduled at or before t'. The jobs scheduled after j_k must be released after t'; otherwise, we could swap the scheduled times of j_k and of such a job, which is feasible because j_k

has the latest deadline among the jobs, thereby resulting in a greater t', a contradiction. In this way, we reach two subproblems with intervals $[t_1, t']$ and $[t'+1, t_2]$. We also know that the number of jobs that must be scheduled in the first and second intervals (excluding j_k itself) are $k-i-1$ and i respectively.

Now we consider two possibilities: (1) $t' < t_2$ and (2) $t' = t_2$. In the first case, we can suppose an optimal solution in which j_k is scheduled on the first processor: if some other job is scheduled on that processor, we can swap its assignment with j_k's. Therefore, at time t', the jobs of the first interval cannot be scheduled on the first processor, which can be represented by setting $q' = 1$ for this subproblem. The subproblem for the second interval simply inherits $q' = q$. In the second case, we can suppose an optimal solution in which j_k is scheduled on the $(q+1)$st processor: again, if some other job is schedule on that processor, we can swap its assignment with j_k's. Thus, at time $t' = t_2$, the jobs cannot be scheduled on the first $q+1$ processors in the subproblem, so we set $q' = q+1$. In this case there is no second subproblem.

For calculating $C_{t_1,t_2,k,q,\ell_1,\ell_2}$, we have four cases: (1) $t' = t_2$; (2) $t' = t_2 - 1$; (3) $t_1 < t' < t_2 - 1$; and (4) $t' = t_1$. In the first case, $C_{t_1,t_2,k,q,\ell_1,\ell_2}$ can be computed directly from $C_{t_1,t_2,k-1,q+1,\ell_1,\ell_2-1}$. In the second case, we need to guess the number ℓ' of processors occupied at time $t' = t_2 - 1$. In this way, the solution can be computed using values $C_{t_1,t_2-1,k-i-1,1,\ell_1,\ell'}$ and $C_{t_2,t_2,i,q,\ell_2,\ell_2}$ for $0 \le \ell' \le p$. In the third case, we need to guess the numbers of processors occupied at times t' and $t'+1$, call them ℓ' and ℓ'' respectively. Thus the solution can be calculated using the values $C_{t_1,t',k-i-1,1,\ell_1,\ell'}$ and $C_{t'+1,t_2,i,q,\ell'',\ell_2}$ for $0 \le \ell', \ell'' \le p$. Finally, in the fourth case, similar to the second case, we can find the solution using the values $C_{t_1,t_1,k-i-1,1,\ell_1,\ell_1}$ and $C_{t_1+1,t_2,i,q,\ell',\ell_2}$ for $0 \le \ell' \le p$.

As in Baptiste's dynamic program [Bap06], we can argue that the number of choices for t_1 and t_2 is polynomial in n. Specifically, Baptiste proved that there is an optimal schedule in which the starting time of any job i is within distance n of some release date or deadline. Hence, t_1, t_2, and t' can have at most n^2 values. Therefore, the size of array C is $n^2 \times n^2 \times n \times p \times p \times p = n^5 p^3$. We use $O(p^2 n^2)$ operations to calculate each entry of the array C. Thus, the total number of operations needed for solving the problem is $O(n^7 p^5)$. □

As mentioned in the introduction, because our algorithm is exact, it essentially solves both the power-minimization and gap-scheduling problems. However, there is a subtle difference, which we now address. In the *multiprocessor power-minimization problem*, a processor is allowed to stay in the active state even during a gap (when no job is scheduled). Thus a gap of length ℓ incurs a cost of $\min\{\ell, \alpha\}$. We show that the polynomial-time dynamic program described above can be adapted to solve this problem as well.

Lemma 2 *Any feasible instance of multiprocessor power minimization has an optimal solution in which, if a processor P_q is in the active state at time t, then all lower-numbered processors $P_1, P_2, \ldots, P_{q-1}$ are also in the active state at time t.*

Proof: The proof is almost identical to the proof of Lemma 1. Instead of counting the gaps, we can simply count the active time units of the processors which the processor is in sleep state in the previous time unit. Suppose that in optimal solution, we have n_1 active processors in time unit t and n_2 active processors in time unit $t+1$ (for an arbitrary t). We know that at least $n_2 - n_1$ gaps will be created here. If we move active time units to the first processors, the number of produced gaps will not be more than this. Therefore, by moving all active time units to the first processors, the total number of gaps will not increase. □

Now using Lemma 2 and because permuting jobs or scheduling a job in another active state of a processor does not change the number of gaps, we conclude that there is an optimal solution in which, for any time unit t, if there are a active processors and $b \leq a$ processors are executing a job during time unit t, then exactly the first b processors are executing a job and the first a processors are in the active state during time unit t. Now the following theorem can be concluded with the new active state.

Theorem 2 *The multiprocessor power minimization problem can be solved in polynomial time even if a processor can be in the active state without executing a job.*

Proof: The proof is similar to the proof of Theorem 1. We define C_{t_1,t_2,k,l,p_1,p_2} as before, except that p_1 and p_2 are no longer the number of occupied processors in time units t_1 and t_2; instead, p_1 and p_2 are the number of processors that are in the active state during time units t_1 and t_2. The rest of the proof is the same. □

3 $(1 + \frac{2}{3}\alpha)$-Approximation for Multi-Interval Power Minimization

In the *multi-interval power minimization problem*, we have n jobs $j_1, j_2, \ldots, j_n$; each job j_i has a unit processing time and a specified set of times T_i at which it can execute. A *feasible schedule* is an assignment of each job j_i to a unique integer time $t_i \in T_i$. A *gap* is a finite maximal interval of time during which no jobs are scheduled. The processor might be in active state even in *gaps*. The *power consumption* of a schedule is the number of time units during which the processor is in active state, plus α times the number of transitions to the active state. Our goal is to determine whether there is a feasible schedule, and if so, to find one that minimizes the power consumption.

Every schedule is within a $1 + \alpha$ factor of optimal, because each job incurs power consumption of either 1 (for execution) or $1 + \alpha$ (for waking up and execution). In Section 4.2, we show that such a dependence on α is necessary. In this section, we reduce the dependence on α:

Theorem 3 *For any constant $\varepsilon > 0$, multi-interval power minimization has a polynomial-time $\left(1 + \left(\frac{2}{3} + \varepsilon\right) \alpha\right)$-approximation algorithm.*

We start with a preliminary result about extending partial schedules which exploits the connection between scheduling and bipartite matching:

Lemma 3 *Given a feasible schedule S' for some subset of $n' \leq n$ jobs that uses g gaps, if there exists a feasible schedule for all n jobs, then we can construct a feasible schedule S for all n jobs that uses at most $g + n - n'$ gaps.*

Proof: We start with schedule S' for the n' and iteratively add additional jobs from the $n - n'$ remaining jobs. We construct a bipartite graph $G = (X \cup Y, E)$ with bipartition (X, Y), where X is the set of jobs and Y is the set of time units. We place an edge $e = (j_i, t) \in E$ precisely when job j_i is allowed to be scheduled at time t, i.e., $t \in T_i$. Our current schedule S corresponds to a matching in graph G that leaves some subset of vertices in X unmatched. The existence of a feasible schedule for all jobs implies the existence of a matching in G that matches every vertex in X. Therefore, the problem of adding another job to the schedule reduces to finding an augmenting path in G, which can be solved in time polynomial in the number n of jobs; see, e.g., [Wes96]. Reversing an augmenting path adds exactly one new execution time to the schedule, thus increasing the number of gaps by at most one. After reversing $n - n'$ such augmenting paths, we reach a feasible schedule S for all n jobs using at most $n - n' + g$ gaps. □

Our next lemma requires some notation. Given a feasible schedule S that performs a subset of some $n' \leq n$ jobs, let T_S denote the set of n' time units during which the jobs are scheduled. Also let $L_{S,k,i} = \{t \mid t \equiv i \pmod{k}$ and $t+m \in T_S$ for all $0 \leq m < k\}$, and let $L'_{S,k,i} = \{t \mid t' \leq t < t'+k$ for some $t' \in L_{S,k,i}\}$. Define a *span* to be a maximal interval of time in which jobs are scheduled (the opposite of a gap). Thus the number of spans is one more than the number of gaps.

Lemma 4 *Suppose that S is a feasible schedule of all n jobs in M spans. For any $k > 1$, there is an i with $0 \leq i < k$ such that $|L_{S,k,i}| \geq \frac{n-M(k-1)}{k}$.*

Proof: For each i with $0 \leq i < k$, let $T_i = T_S \setminus L'_{S,k,i}$. Consider an arbitrary span of S, $I = [t_1, t_2]$. We prove that average size of $I \cap T_i$ for $0 \leq i < k$ is at most $k-1$. We consider three cases: (1) $|I| < k$; (2) $k \leq |I| \leq 2k-2$; and (3) $|I| > 2k-2$. In the first case, $|I| < k$, the claim is trivial.

In the second case, $|I| \leq 2k-2$, so for $|I|-k+1$ values of i, $|I \cap T_i| = |I|-k$, while for the other values $|I \cap T_i| = |I|$. Hence the average size of T_i is $\Big[(|I|-k+1)(|I|-k) + (2k-|I|-1)\,|I|\Big]/k = \frac{k^2-k}{k} = k-1$.

In the third case, $|I| > 2k-2$, so at least one member of $L'_{S,k,i}$ appears in I. Thus, for any i with $0 \leq i < k$, $|I \cap T_i|$ is strictly less than $|I|$. The time units of I that are in T_i form two time intervals $[t_1, t_1+a_i-1]$ and $[t_2-b_i+1, t_2]$ where a_i and b_i are nonnegative integers and $|T_i \cap I| = a_i + b_i$. By definition, we have $a_{i+1} \equiv a_i + 1 \pmod{k}$ and $b_{i+1} \equiv b_i - 1 \pmod{k}$. Therefore, $\sum_{i=0}^{k-1} a_i = \frac{k(k-1)}{2}$ and $\sum_{i=0}^{k-1} b_i = \frac{k(k-1)}{2}$. We conclude that the average size of $|I \cap T_i| = a_i + b_i$ is $k-1$.

Because we have M spans, the average size of T_i is at most $M(k-1)$ and there exists an $0 \leq i < k$ such that $|T_i| \leq M(k-1)$. Thus we have $L'_{S,k,i} = n - T_i \geq n - M(k-1)$ and $L_{S,k,i} \geq \frac{n-M(k-1)}{k}$. □

Next we show a connection to *k-set packing*. In this problem, we are given a collection C of subsets of an underlying base set S, with the property that each set in C has cardinality at least k, where $k \geq 3$. A *set packing* is a subcollection $C' \subseteq C$ of disjoint sets: $X \cap Y = \emptyset$ for $X, Y \in C'$. The goal is to find a set packing of maximum cardinality. Hurkens and Shrijver [HS89] developed a $(\frac{k}{2} + \varepsilon)$-approximation algorithm for the k-set packing problem, for any $\varepsilon > 0$.

Lemma 5 *Suppose that there exists a feasible schedule S of all n jobs in M spans. For any constant $k > 1$ and any $\varepsilon > 0$, there is a polynomial-time algorithm that schedules $\big(n - M(k-1)\big) \cdot \left(\frac{2}{k+1} - \varepsilon\right)$ jobs in $\frac{n-M(k-1)}{k}\left(\frac{2}{k+1} - \varepsilon\right) + 1$ spans.*

Proof: For any i with $0 \leq i < k$, we construct an instance of $(k+1)$-set packing problem as follows. The underlying base set S is $\{j_1, j_2, \ldots, j_n\} \cup \{t \mid t \equiv i \pmod{k}\}$. The collection C is defined as follows: if $j_{a_0}, j_{a_2}, \ldots, j_{a_{k-1}}$ is a sequence of jobs, and if there exists a time $t \in S$ such that, for any l with $0 \leq l < k$, j_{a_l} can be scheduled at time $t+l$, then we insert set $\{j_{a_0}, j_{a_1}, \ldots, j_{a_{k-1}}, t\}$ into the collection C. By Lemma 4, for some i, this instance of the $(k+1)$-set packing problem has a set packing of size at least $\frac{n-M(k-1)}{k}$. The polynomial-time approximation algorithm [HS89] gives us a set packing of size at least $A = \frac{n-M(k-1)}{k}\left(\frac{2}{k+1} - \varepsilon\right)$. Therefore, we can schedule $k\,A$ jobs in these A intervals, implying at most A gaps and thus at most $A+1$ spans. □

Combining Lemmas 3 and 5, we obtain the following result:

Corollary 1 *Suppose that there exists a feasible schedule S of all n jobs in M spans. For any constant $k > 1$ and any $\varepsilon > 0$, there is a polynomial-time algorithm that schedules all jobs in $n - \left(\frac{n-M(k-1)}{k}\right) \cdot \left(\frac{2}{k+1} - \varepsilon\right) \cdot (k-1) + 1$ spans.*

We are now ready to conclude a polynomial-time $\left(1+\left(\frac{2}{3}+\varepsilon\right)\alpha\right)$-approximation algorithm:

Proof of Theorem 3: Suppose that the optimal solution has M spans. For simplicity, we assume that the last job also incurs a cost of α to return the system to the sleep state; this assumption can be avoided by a suitable tweaking of ε.

By Corollary 1, if we let $k=2$, we can schedule all jobs in $\left(\frac{2}{3}+\varepsilon\right)n+\left(\frac{1}{3}-\varepsilon\right)M$ spans. Thus the power consumption is at most $n+\left(\left(\frac{2}{3}+\varepsilon\right)n+\left(\frac{1}{3}-\varepsilon\right)M\right)\alpha$ if we keep processor in the active state precisely when it is doing a job. Now we consider two cases to compute the optimal cost. First, if $\alpha \le 1$, then it is cost effective to keep processor in the sleep state between spans. Thus the optimal solution has a power consumption of $n+M\,\alpha$. Because both fractions $\frac{n+\left(\frac{2}{3}+\varepsilon\right)n\,\alpha}{n}$ and $\frac{\left(\frac{1}{3}-\varepsilon\right)M\alpha}{M\alpha}$ are at most $1+\left(\frac{2}{3}+\varepsilon\right)\alpha$, the upper bound on the approximation factor $\frac{n+\left(\left(\frac{2}{3}+\varepsilon\right)n+\left(\frac{1}{3}-\varepsilon\right)M\right)\alpha}{n+M\,\alpha}$ is also at most $1+\left(\frac{2}{3}+\varepsilon\right)\alpha$. On the other hand, if $\alpha > 1$, then the optimal solution has a power consumption of at least $n+M$. We know that $\frac{n}{n+M} \le 1$ and $\frac{\left(\left(\frac{2}{3}+\varepsilon\right)n+\left(\frac{1}{3}-\varepsilon\right)M\right)\alpha}{n+M} \le \left(\frac{2}{3}+\varepsilon\right)\alpha$. Thus the approximation factor is at most $1+\left(\frac{2}{3}+\varepsilon\right)\alpha$. □

4 Hardness of Approximation for Multi-Interval Power Minimization

In this section we prove inapproximability results for multi-interval power minimization (as defined in Section 3). Our results are based on reductions from versions of set cover, and assume only that P $\neq$ NP.

4.1 $\Omega(\lg n)$ Hardness

First we prove that, if the transition cost α is part of the input, then the problem is set-cover hard:

Theorem 4 *Multi-interval power minimization has no polynomial-time $o(\lg n)$-approximation algorithm unless* P = NP.

Proof: We give an approximation-preserving reduction from set cover, which is not $o(\lg n)$-approximable unless P = NP [Fei98]. Let (E, C) be an instance of set cover, where $E=\{e_1, e_2, \ldots, e_n\}$ is the universe of elements and $C=\{c_1, c_2, \ldots, c_s\}$ is a collection of subsets of E. We build an instance of multi-interval power minimization as follows. For each set $c_i \in C$, construct an interval I_i of length $|c_i|$. Construct these intervals so that the distance between any two of them is larger than n^3. For each element $e_i \in E$, define a job j_i that is allowed to be executed during any interval I_k for which $e_i \in c_k$. Also construct one interval I_{s+1} of size 1 and a job j_{s+1} that is allowed to be executed only during I_{s+1}. Define $\alpha = n$.

A set cover S of size k is easy to convert into a solution to multi-interval power minimization with cost $(1+k)\,n$. For each job, assign it to the interval I_i corresponding to a set $c_i \in S$. (By definition of set cover, such a set always exists.) Within each interval I_i, execute the assigned jobs consecutively. (Because the length of interval I_i is $|c_i|$, there is enough time to execute the at most $|c_i|$ assigned jobs.) The number of spans is thus k plus one for the extra interval I', so the number of gaps is k, for a power consumption of $n+k\,\alpha = n+k\,n$.

Conversely, a solution to multi-interval power minimization with cost $(1+k)\,n$ can also be converted into a set cover of size at most k. If $k \ge n$, such a set cover is trivial: pick any set for each element in E. Otherwise, we construct a set cover S consisting of each set c_i whose corresponding interval I_i executes at least one job. The processor cannot stay awake between two intervals, because the $> n^3$ distance would incur a power consumption of more than $n^3 > (1+k)\,n$ (assuming $n \geq 2$). Thus the number of spans is at least the number of intervals that execute at least one job, counting I', which is $|S|+1$. Hence the number of gaps is at least $|S|$, so the power consumption is at least $n + |S|\,\alpha = n + |S|\,n$. But we supposed that the power consumption is $(1+k)\,n$, so $(1+k)\,n \geq n + |S|\,n$, i.e., $|S| \leq k$.

Because these transformations scale uniformly by a factor of n (other than the negligible ± 1), a $o(\lg n)$-approximation for multi-interval power minimization would imply a $o(\lg n)$-approximation for set cover, and thus P = NP. □

4.2 $\Omega(\lg \alpha)$ Hardness

Next we prove that the approximation factor must depend on α at least logarithmically:

Theorem 5 *Multi-interval power minimization with transition cost α has no polynomial-time $o(\lg \alpha)$-approximation algorithm unless* P = NP.

Proof: The reduction is similar to that of Theorem 4. In this case, however, the source problem is a restriction of set cover, *B-set cover*, where every set c_i has size at most B. This problem is not $\varepsilon \lg B$-approximable for some $\varepsilon > 0$ assuming P $\neq$ NP [Tre01]. The only difference in our reduction is that we let $\alpha = B$ instead of n. As before, we can show that there is a set cover of size k if and only if the constructed instance of multi-interval power minimization has a schedule with power consumption $n + k\,\alpha = n + k\,B$.

Suppose that the optimal schedule has a power consumption of $n + k\,B$. Because each set has size at most B, the optimal set cover has size at least n/B, and hence $k \geq n/B$. Thus $k\,B \geq n$.

Now, if we had an $(\frac{1}{2}\varepsilon \lg B)$-approximation for multi-interval power minimization, we would obtain a schedule with power consumption at most $(n + k\,B)\,\frac{1}{2}\varepsilon \lg B$. Because $k\,B \geq n$, this approximate power consumption is at most $n + k\,B\,\varepsilon \lg B$. Using the equivalence, we obtain a set cover of size $k\,\varepsilon \lg B$. In other words, we obtain a $(\varepsilon \lg B)$-approximation to set cover, which for sufficiently small $\varepsilon > 0$ implies that P = NP. □

5 Hardness of Approximation for Gap Scheduling

In the *multi-interval gap scheduling problem*, we have n jobs $j_1, j_2, \ldots, j_n$; each job j_i has a unit processing time and a specified set of times T_i at which it can execute. A *feasible schedule* is an assignment of each job j_i to a unique integer time $t_i \in T_i$. A *gap* is a finite maximal interval of time during which no jobs are scheduled. Our goal is to find a feasible schedule that minimizes the number of gaps.

For simplicity, we define one of the infinite intervals to be a gap as well. This change does not change the optimal solution, and has a negligible impact on approximation.

5.1 $\Omega(\lg n)$ Hardness for 2-Interval Gap Scheduling

To get started, we prove that (general) gap scheduling is set-cover hard:

Theorem 6 *Multi-interval gap scheduling has no polynomial-time $o(\lg n)$-approximation algorithm unless* P = NP.

Proof: This result follows by a simple adaptation of the proof of Theorem 4. We can use the same reduction from set cover. Before we showed that there is a set cover of size k if and only if the constructed instance has a power consumption of $n + k\,\alpha$. But such a power consumption implies having exactly k gaps. Thus there is a set cover of size k if and only if there is a feasible schedule with k gaps. Therefore, a $o(\lg n)$-approximation for multi-interval gap scheduling would imply a $o(\lg n)$-approximation for set cover, and thus P = NP. $\square$

Corollary 2 *It is* NP*-hard to approximate multi-interval gap scheduling within a* $o(\lg N)$ *factor, where* N *is the size of input.*

Proof: According to [Fei98], it is NP-hard to approximate set cover within a $o(\lg n)$ factor even when the input size is bounded from above by a polynomial of the number of elements. Thus, approximating set cover within a $o(\lg N)$ factor is also NP-hard where N is the size of input. Using the reduction in Theorem 6, we conclude the hardness result for multi-interval gap scheduling. $\square$

Theorem 7 *It is* NP*-hard to approximate* 2*-interval gap scheduling within a* $o(\lg N)$ *factor, where* N *is the size of input.*

Proof: We give an approximation-preserving reduction from multi-interval gap scheduling to 2-interval gap scheduling. For an arbitrary job j in a given instance of multi-interval gap scheduling, if the number of intervals assigned to j is greater than two, we can replace j by some new jobs each executable in exactly two intervals.

Suppose that a given job j can be executed in k intervals. We assign a new interval to this job, called an extra interval, whose length is $2k-1$. We create k dummy jobs such that ith dummy job could only be executed in the $2i-1$-st unit of extra interval. Additionally, for each interval I_i $(1 \le i \le k)$ assigned to j, we add job r_i which could be executed either in I_i or in extra interval. We put the extra intervals related to all jobs consecutively such that no gap may be formed between them. In this way, we produce an instance of 2-interval gap scheduling because neither dummy jobs nor r_is have more than two intervals to be executed in.

There is an optimal solution in the presented construction such that extra interval is completely occupied; no gap is within extra interval. If there exists a free unit in extra interval, there must be at least one job which could run in that unit. By moving this job to that position, it fills the space between two dummy jobs; therefore, the overall number of gaps will not increase. Iterating the process can fill all extra intervals completely.

If all extra intervals are completely filled, it means that exactly one of the r_is assigned to each j is out of extra interval. This r_i must be executed in the corresponding I_i. Thus, an algorithm selects the appropriate I_i for each job and performs the job somewhere in it; a selected job related to j can be executed in all places which j could. Therefore, the jobs which are out of extra interval would completely resemble the instance of multi-interval gap scheduling. Thus, the solution for this instance of 2-interval gap scheduling has exactly one more gap than the related instance of gap scheduling because extra interval itself creates a gap. We remark that all extra intervals come consecutively and there is no free space between them.

To avoid the excessive gap which comes from extra interval, we try to put extra interval exactly after the last occupied unit. Although we do not know the exact positions where a job will occupy in the optimal solution, we can guess that by trying all possible positions. In this way, the excessive gap will be destroyed and the solution for 2-interval gap scheduling will be exactly the same as the one for multi-interval gap scheduling. In our construction, the input size of 2-interval gap scheduling problem is bounded above by a polynomial of the input size of multi-interval gap scheduling. We conclude that the logarithm of input sizes of these two instances are of the same order. Therefore, 2-interval gap scheduling cannot be approximated within a $o(\lg N)$ factor. □

5.2 $\Omega(\lg n)$ Hardness for 3-Unit Gap Scheduling

Theorem 8 *It is* NP*-hard to approximate 3-unit gap within* $o(\lg N)$ *factor, where* N *is the size of input.*

Proof: We show that 3-unit gap can solve multi-interval gap scheduling to conclude that 3-unit gap cannot be approximated within $o(\lg N)$ factor. For an arbitrary job j in a given instance of multi-interval gap scheduling, if the number of time units assigned to j is greater than three, we replace j by some new jobs such that each job can be executed in at most three time units.

Suppose that a given job j can be executed in k units say $t_1, \ldots, t_k$. We assign a new interval to this job called extra-interval whose length is $2k-1$. We create k dummy jobs such that ith dummy job can only be executed in the $2i-1$-st unit of extra-interval. For each unit t_i where $1 \leq i \leq k-1$, we add a job j_i which could be run in the $2i$th or $((2i+2) \bmod 2k)$th unit of extra-interval or in t_i. We also add a job j_k which could be run in t_k or in the second or fourth place of extra-interval. We put all extra-intervals consecutively, thus, no gap will be formed between them. According to the construction, each j_i can be run in exactly 3 units. Therefore, if we replace j by j_is for all jobs, we reach an instance of 3-unit gap.

According to the above construction, every combination of $k-1$ jobs $j_{a_1}, j_{a_2}, \ldots, j_{a_{k-1}}$ could be scheduled to fill extra-interval completely. If $\{a_1, a_2, \ldots, a_{k-1}\} = \{1, 2, \ldots, k-1\}$, then we perform job j_i in the $2i$th unit of extra-interval. Otherwise, suppose that we want to run j_k instead of another job say j_q. If $q = 1$ or $q = 2$, we can schedule j_k exactly in its place. Otherwise, for each job j_i where $1 \leq i \leq q-1$, perform it in $(2i+2)$nd unit of extra-interval. In this way, $2q$th place of extra-interval will be filled and the second unit in extra-interval will be free. Now, we schedule j_k in the second unit. Therefore, extra-interval can be filled with every set of $k-1$ jobs.

There is an optimal solution in the presented construction such that extra-interval is completely occupied; no gap is within extra-interval. If there exists a free unit in extra-interval, using the previous statement, there must be at least one job which could run in that unit. By moving this job to that position, it fills the space between two dummy jobs; therefore, the overall number of gaps will not increase. Iterating the process can fill all extra-intervals completely.

If all extra-intervals are completely filled, it means that exactly one of the j_is assigned to each j is out of extra-interval. This j_i must be executed in the correspondent t_i. Thus, an algorithm can select the appropriate j_i for each job j; Therefore, a selected job related to j can be executed in all places which j could. In this way the jobs which are out of extra-interval would completely resemble the multi-interval gap scheduling instance. The solution for this instance of 3-unit gap has exactly one more gap than the related instance of multi-interval gap scheduling because extra-interval itself creates a gap. We remark that all extra intervals come consecutively and there is no free space between them.

To avoid the excessive gap which comes from extra-interval, we try to put extra-interval exactly after the last occupied unit. Although we do not know the exact positions where a job will occupy in the optimal solution, we can guess that by trying all possible positions. In this way, the excessive gap will be destroyed and the solution for 3-unit gap will be exactly the same as the one for multi-interval gap scheduling. If we create additional job j_i only for necessary units (the units which an optimal solution may use them), according to [Bap06, Prop. 2.1] the input size of 3-unit gap problem is bounded from above by a polynomial of the input size of multi-interval gap scheduling problem. We conclude that the logarithm of input sizes of these two instances are of the same order. Therefore, the 3-unit gap problem also could not be approximated within the $o(\lg N)$ factor. $\square$

5.3 $\Omega(1)$ Hardness for 2-Unit and 1-Unit Gap Scheduling

A *gap'* is a sequence of consecutive time units in which no job interval exists, and consequently, no job can be scheduled. We can suppose that, in the 2-unit and 1-unit gap scheduling problems, each gap' has one time unit. Otherwise, we could reduce the gap' to one time unit, and because no job can be scheduled in a gap', reducing it does not change the problem.

Theorem 9 *A polynomial-time c-approximation algorithm for disjoint-unit gap scheduling yields a polynomial-time $(c+\varepsilon)$-approximation algorithm for two-unit gap scheduling; and a polynomial-time c-approximation algorithm for two-unit gap scheduling yields a polynomial-time $(c+\varepsilon)$-approximation algorithm for disjoint-unit gap scheduling.*

Proof: Consider an instance of two-unit gap scheduling with jobs $J_1, J_2, \ldots, J_n$. For $1 \le i \le n$, let T_i be the set of units during which job J_i can be scheduled, with $|T_i| \le 2$. We construct a bipartite graph $G(X,Y)$ which X is the set of jobs and $Y = \bigcup_{i=1}^{n} T_i$. There is an edge between job J_i and time unit t if and only if $t \in T_i$. We can schedule the jobs for different connected components of graph G independently. Consider a connected component $H(X',Y')$ of G which contains jobs $X' = \{J_{a_1}, J_{a_2}, \ldots, J_{a_k}\}$ and time units $Y' = \{t_1, t_2, \ldots, t_{k'}\}$. Let E' be the number of edges of H. We know that $|T_{a_i}| \le 2$, thus, $|E'| \le 2k$. On the other hand, connectivity of H implies that $k + k' - 1 \le |E'|$, consequently, $k' \le k+1$. Because there exists a valid solution, all jobs of set X' must be scheduled only in time units of Y' which implies that $k \le k'$. Therefore, k' is equal to k or $k+1$. If $k' = k$, it makes no difference how to schedule the jobs of set X', because all time units will be used anyway. In the other case, where $k' = k+1$, any scheduling leaves system idle in exactly one of the time units $Y' = \{t_1, t_2, \ldots, t_{k'}\}$. We prove that, for any i with $1 \le i \le k+1$, all k jobs can be scheduled in time units $Y' - \{t_i\}$. We know that all jobs of set X' can be scheduled. Without loss of generality, suppose that these jobs can be scheduled in time units $t_1, t_2, \ldots, t_k$ which means that there exists a matching M between vertices X' and $Y' - t_{k+1}$. We want to prove that the jobs can be scheduled in time units $Y' - \{t_i\}$ $(1 \le i \le k)$. There is a path P between t_i and t_{k+1} because t_i and t_{k+1} are in the same connected component. According to the fact that vertices of part X of this path have at most two neighbors, they occur in this path by their both edges which one of them is in the matching M and the other one is not. Therefore, path P is an alternating path of matching M. Switching the edges of this alternating path gives us a matching that saturates vertex t_{k+1} and does not saturate vertex t_i. Thus, we can schedule all jobs of set X' in time units $Y' - \{t_i\}$. We construct our instance of disjoint-unit gap scheduling problem as follows. For connected components $H(X',Y')$ where $|Y'| = |X'| + 1$, we put a job which can be done in time units of the set Y'. For any gap' $I = \{t\}$, we put a job which can be scheduled only in time unit t.

For any scheduling of the jobs in the new instance of the disjoint-unit gap scheduling problem, there is a scheduling of jobs for the two-unit gap scheduling problem such that the state of the system is reversed for all time units. Thus, the number of gaps in these two instances are almost equal (differ by at most 1). This difference can create a 1/OPT additive difference in approximation factors of these two problems, where OPT is the number of gaps in the optimum solution of the two-unit gap scheduling problem. Because the problem can be solved in polynomial time for small values of OPT, the difference can be reduced to an arbitrarily small value ε.

Now consider an instance of disjoint-unit gap scheduling with jobs $J_1, J_2, \ldots, J_n$. For $1 \leq i \leq n$, let T_i be the set of units during which job J_i can be scheduled, with $|T_i \cap T_j| = 0$. For every job j_i with $|T_i| = k > 1$, we put $k-1$ jobs. Suppose $T = \{t_1, t_2, \ldots, t_k\}$. The mth job can be done in t_m and t_{m+1}, for all $1 \leq m \leq k-1$. For any gap' $I = \{t\}$, we put a job that can be scheduled only in time unit t. Similarly we can can show that the optimal solutions of these two instances are almost equal (differ by at most one).

Construction of the new instances can be done in polynomial time because of the specific constraints on the two-unit gap scheduling and the disjoint-unit gap scheduling instances. Thus approximation factors of problem two-unit gap scheduling and disjoint-unit gap scheduling differ by at most ε. □

Using B-set cover, we show that disjoint-unit gap scheduling has no constant-factor approximation:

Theorem 10 *It is* NP*-hard to approximate disjoint-unit gap scheduling within any constant factor c.*

Proof: We prove that, if disjoint-unit gap scheduling can be approximated within a factor of c, then B-set cover can also be approximated within the same factor, for any constant B. Let (E, C) be an instance of B-set cover, where $E = \{e_1, e_2, \ldots, e_m\}$ is the universe of elements, and $C = \{c_1, c_2, \ldots, c_s\}$ is a set of subsets of E each of which has at most B elements. We convert the B-set-cover instance into an instance of disjoint-unit gap scheduling. For each subset $A \subseteq c_i$ where $1 \leq i \leq s$, let l be an interval whose length is equal to $|A|$, and suppose no two intervals like A overlap. Consider a job j_i for each element $e_i \in E$. Let j_i be executed in jth unit of interval l if e_i is the jth smallest element of A. Because B is constant, the number of subsets of each $c_i \in C$ is constant, so the size of above instance of disjoint-unit gap scheduling is bounded from above by a polynomial with respect to the size of B-set-cover instance. Next we show that, with the provided construction, disjoint-unit gap scheduling is as hard as B-set cover.

The solution of B-set-cover instance could be changed into a solution with the same size for disjoint-unit gap scheduling as follows; in a solution of B-set cover, we choose some sets of C such that each element of the universe can be assigned to a chosen set. Suppose set c_i is chosen and $A \subseteq c_i$ is the set of assigned elements to c_i. In disjoint-unit gap scheduling instance, we schedule the correspondent jobs to elements of A exactly in the interval which we considered for A. Thus, all jobs are scheduled and the number of gaps is equal to the solution of the B-set-cover instance. On the other hand, a solution for disjoint-unit gap scheduling could also be converted to a valid one for the B-set-cover instance; select a set if the correspondent interval of at least one of its subsets contains some jobs. Because all intervals are disjoint, the number of gaps required for performing all jobs cannot be less than the number of intervals with at least one execution. Hence, a valid solution for disjoint-unit gap scheduling can create a solution of the same or smaller size for the set-cover problem; consequently, the optimal solution for disjoint-unit gap scheduling can be converted to the optimal solution for the set-cover instance. □

Combining Theorems 9 and 10, we obtain the following result.

Corollary 3 *It is* NP*-hard to approximate two-unit gap scheduling within any constant factor c.*

6 $O(\sqrt{n})$-Approximation for Maximizing Throughput for Given Gap Bound

The *minimum-restart problem* is a variation of multi-interval gap scheduling. As usual, we have n jobs $j_1, j_2, \ldots, j_n$, and each job j_i has a unit processing time and a specified set of times T_i at which it can execute. We are also given a bound k on the number of gaps. Our goal is to find a feasible schedule for the maximum number of jobs subject to having at most k gaps.

Theorem 11 *There is an polynomial-time $O(\sqrt{n})$-approximation algorithm for the minimum-restart problem.*

Proof: The approximation algorithm is greedy with k steps. In each step, we select the biggest time interval $[a, b]$ such that $b - a + 1$ unscheduled jobs can be scheduled in it without any collision. We can find such interval using a maximum matching algorithm in each iteration. Then we schedule those $b - a + 1$ jobs in it.

A working interval is a pair (I, X) where I is a time interval with size l and X is a set of l jobs which can be scheduled in I without any collision. Suppose our algorithm chooses working intervals $A = \{A_1 = (I_1, X_1), A_2 = (I_2, X_2), \ldots, A_k = (I_k, X_2)\}$ respectively (A_1 at first iteration, then A_2 and so on). Let $Y = \{B_1, B_2, \ldots, B_k\}$ be the set of working intervals of the optimum solution. We say two working intervals overlap if and only if either their time intervals overlap or intersection of their set of jobs is not empty.

Now we prove that the number of jobs scheduled in optimum solution is no more than $2\sqrt{n}$ times of the number of scheduled jobs in our solution. For any i, if $X_i > \sqrt{n}$ then we have scheduled at least $\sqrt{n}$ jobs and our algorithm would be an $O(\sqrt{n})$-approximation. In the other case, let Y_i be the set of working intervals in optimum solution which overlap with A_i and do not overlap with any $A_{i'}$ for $1 \leq i' < i$. We know that the number of scheduled jobs in any working interval of Y_i is at most the number of scheduled jobs in A_i otherwise we could choose that working interval instead of A_i in our greedy algorithm. According to the fact that working intervals of Y_i are disjoint and all of them overlap with A_i, $|Y_i|$ is at most $|I_i| + |X_i| = 2|X_i|$ because each time unit or job of A_i can be used in at most one working interval of Y_i. Consequently, the number of scheduled jobs using working intervals of Y_i is at most $|Y_i||A_i| \leq 2|X_i||A_i| \leq 2\sqrt{n}|A_i|$ which means that, for any i with $1 \leq i \leq k$, the number of scheduled jobs using working intervals of Y_i is at most $O(\sqrt{n})$ times of the number of scheduled jobs in A_i.

Let Y' be the set of working intervals in Y that do not overlap with any A_i with $1 \leq i \leq k$, and let A' be the set of working intervals such as A_i such that Y_i is empty. As we know, the number of working intervals in Y is equal to the number of working intervals in A, thus, $|Y'|$ could not be more than $|A'|$. According to the greedy algorithm, the number of scheduled jobs in any working interval of A' is not less than the number of scheduled jobs in any working interval of Y' because the working intervals of Y' do not overlap with any working interval of A'. Therefore, the number of scheduled jobs using working intervals of A' is not less than the number of scheduled jobs of Y'. We conclude that the number of all scheduled jobs in our solution is not less than $O(\sqrt{n})$ times of the number of scheduled jobs in optimum solution. □

Acknowledgments

We thank the anonymous referees for helpful comments.

References

[AIS04] John Augustine, Sandy Irani, and Chaitanya Swamy. Optimal power-down strategies. In *Proceedings of the 45th Symposium on Foundations of Computer Science*, pages 530–539, Rome, Italy, October 2004.

[Bap06] Philippe Baptiste. Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 364–367, Miami, Florida, 2006.

[BBM00] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, 2000.

[Fei98] Uriel Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.

[FHKN06] Uri Feige, MohammadTaghi Hajiaghayi, Sanjeev Khanna, and Seffi Naor. On approximation of minimum-gap scheduling. Unpublished manuscript, 2006.

[HS89] C. A. J. Hurkens and A. Schrijver. On the size of systems of sets every t of which have an SDR, with an application to the worst-case ratio of heuristics for packing problems. *SIAM Journal on Discrete Mathematics*, 2(1):68–72, 1989.

[IP05] Sandy Irani and Kirk R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36(2):63–76, 2005.

[ISG03] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 37–46, Baltimore, Maryland, 2003.

[Tre01] Luca Trevisan. Non-approximability results for optimization problems on bounded degree instances. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, pages 453–461, 2001.

[Wes96] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall Inc., Upper Saddle River, NJ, 1996.