Department of Computer Science
George Mason University
Technical Report Series

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/
703-993-1530

Extra Buffer Resources Improving Competitiveness in Minimizing Energy Consumption

Zhi Zhang
zzhang8@cs.gmu.edu

Fei Li
lifei@cs.gmu.edu

Technical Report GMU-CS-TR-2010-6

Abstract

In this paper, we consider energy management algorithms for scheduling jobs in power-scare scenarios such as embedded computer systems and sensor networks. We focus on investigating the impact of buffer resources in minimizing the total energy cost in an online setting. The online algorithms do not have any assumptions on job arrivals; their worst-case performance is measured in term of competitive ratio, when they are compared with the optimal algorithms with clairvoyance. We prove that with appropriate extra buffer space, an online algorithm can beat an weak optimal offline algorithm in terms of the total energy required. Our research result helps to quantitatively estimate the optimal on-chip buffer resources allocated in real-time systems with power constraints. We also present the lower bound of competitive ratio that any deterministic online algorithm cannot achieve.

1 Introduction

In designing a System-on-Chip (SoC) or a Network-on-Chip (NoC) for embedded computer systems and sensor networks, silicon area and power consumption are most two significant factors to optimize. The techniques in minimizing chip areas are relatively mature. However, the energy consumption has been widely known to become the principal factor of improving system performance. In these power-scarce settings, the benefit of completing a job is usually diluted by its energy expense. Thus, how to develop efficient algorithms to economically manage energy expenditure during executing jobs is a challenging problem. This research topic has attracted much attention recently; see [9, 7] and the references therein. The main algorithmic techniques for reducing energy consumption can be broadly classified in two categories: *dynamic power management (DPM)* and *dynamic voltage scaling (DVS)*. DVS is a technique used in modern microprocessors operated by battery. Voltage and frequency levels of the microprocessor are set properly to meet jobs' performance requirements while energy consumption is minimized. This technique is initialized by Yao et al. [14]. In contrast with DVS, DPM saves energy by putting the system into a lower power state during the *idle periods*, defined as the duration in which there are no pending jobs for the machine to execute. Many dynamic power management strategies — both offline and online solutions such as those in [8, 2, 1, 5] — have been proposed to save energy when the idle periods are long enough. Our research in this paper lies in the line of the research on DPM.

Let us consider Figure 1 as an example to illustrate the packet switch mechanism within an embedded sensor. The system has a few input ports that are accepting packets from its neighbors in the network. These packets are forwarded to other neighbors via the output ports. In general, some buffer slots are allocated to accommodate the packets before they are sent out. However, the buffer space is a kind of precious resource for SoC and NoC. People have studied the on-chip buffer constraint for streaming applications (see [12] and the references therein). However, little work has been done on evaluating the on-chip buffer effect for energy management. In this paper, we try to answer the following two questions that are naturally raised: (1) How do we quantitatively evaluate the effect of the buffer space in serving streaming (online) applications? (2) In serving streaming applications, what is the best tradeoff between the buffer space and the energy required? That is, if we increase the buffer space a little, can we improve the efficiency of power consumption significantly?

One concern on energy consumption comes naturally from the consideration on jobs' delays in execution when they are buffered for some time before being served. In the previous work [8, 1, 5], jobs are supposed to be executed immediately at the time when there is no other job being processed. This implies that in their settings, the 'machine' is non-idling and all jobs have their slacks (to be formally defined in the following) implicitly assumed 0. However, we realize that if jobs are allowed to be postponed in execution (that is, we increase their slacks such that they can be accommodated in the buffer longer before being served), an online algorithm's performance of *weak competitiveness* (defined in Subsection 2) can be improved significantly. The underlying reason is that the energy cost associated with powering on the machine is considerably decreased. To reveal the impacts of jobs' delay in execution on an online algorithm's performance, we study a job's *slack*, defined as the upper bound of the difference between the job's release time and the latest time it should be executed. Our research concludes that the more slacks jobs have, the better competitive ratio an online algorithm can achieve. We note that there exists correspondence between the buffer space required and job slacks: The buffer space should be a constant factor of the slacks of jobs. Thus, the more buffer space is, the better competitiveness an online algorithm has. The rigorous analysis in this paper validate our intuition.

In this paper, motivated by scheduling job in embedded computer systems and sensor networks, we consider the above model of energy management and we call it a *slack model*. Our model is described in Section 2. Note that scheduling jobs is essentially an online decision-making problem, we design a competitive online algorithm for the slack problem in Section 3 and analyze its performance in terms of both *asymptotic running time* and *weak competitive ratio* in Section 4, respectively. Conclusion remarks and related work are described in Section 5.

2 The Slack Model

We design algorithms to schedule unit-length jobs (also called jobs). In our setting, time is discrete and each time interval $[t,\ t+1)$ is called a time step t. We consider the slack model in which online algorithms have no knowledge about future released jobs.

The machine

There is one machine (for instance, a job transmitter in a wired/wireless communication system). At any time, at most one job can be run on the machine. The machine has only two states: `active` and `sleep`. If the machine is currently running a job, it must be at its `active` state and a machine cannot run a job at all at its `sleep` state. When the machine does not run any job (though there *may* exist pending jobs), it can be at either its `active` or `sleep` state. We call the machine *spinning* when it is at the `active` state but no job is running. When it is at the `active` and `sleep` states, the machine consumes energy $\mu \in \mathbb{R}^+$ and $e(s) \in \mathbb{R}^+$ per time unit respectively. Without loss of generality, we assume $\mu > 0$ and $e(s) = 0$.

In order to power on (respectively, off) the machine from a `sleep` (respectively, an `active`) state to an `active` (respectively, a `sleep`) state, we have to pay transition energy $C(s/a) \in \mathbb{R}^+$ (respectively, $C(a/s) \in \mathbb{R}^+$). We denote $C = C(s/a) + C(a/s)$. For unit-length jobs (jobs), the machine can finish processing a job completely in each single time step.

Jobs

Let the set of jobs released be $\mathscr{I}$. Each job $j \in \mathscr{I}$ is released at an integer time $r_j \in \mathbb{Z}^+$. Without loss of generality, for this model, we can assume that in each time step, at most one unit-length job arrives. The same assumption has been made as well in all of the models considered in [10, 11, 8, 2, 1, 5]. (If x jobs arrive at the same time, we simply assume that they are release at time $t,\ t+1,\ t+2,\ \ldots,\ t+x-1$, respectively.) With such an assumption, any schedule can finish all released jobs as long as it processes a job immediately at its arrival. All released jobs are to be finished and $\mathbf{S} = \mathscr{I}$. The extended work (algorithms, their analysis and performance) on scheduling jobs with various lengths are quite similar.

Jobs have slacks κ such that a job has to be executed some time within κ time units after it is released. For offline algorithms, κ is assumed 0. In this paper, we evaluate an online algorithm's performance when its κ is allowed to be positive.

We note here that jobs have no deadlines in the slack model; though we can regard that each job j has a deadline $r_j + \kappa$, where $\kappa \geq 0$ for online algorithms and $\kappa = 0$ for offline algorithms.

Energy cost

Let E denote the total energy spent by the machine in finishing a set of jobs $\mathbf{S} \subseteq \mathscr{I}$. Let $T(a)$ (respectively, $T(s)$)

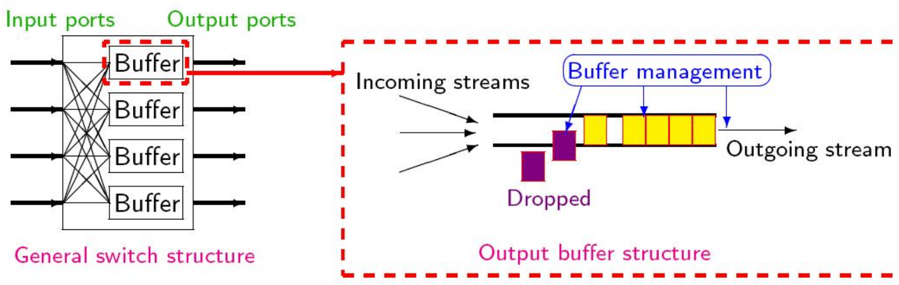

Figure 1: Packet-switching mechanism in an embedded sensor

denote the total amount of time the machine remains at the `active` (respectively, `sleep`) state. Let $m \in \mathbb{Z}^+$ (an integer ≥ 1) be the number of times that the machine is powered on along the course of scheduling jobs. The machine is assumed at its `sleep` state initially and finally. The total energy cost is defined as

$$\begin{aligned} E &= \mu \cdot T(a) + e(s) \cdot T(s) + (C(s/a) + C(a/s)) \cdot m \\ &= \mu \cdot T(a) + C \cdot m. \end{aligned}$$

In the slack model, we note that all released jobs should be completed by the online algorithms as well. Our objective is to minimize the total energy cost E that we have to pay to complete all jobs. For online algorithms, jobs have slacks $\kappa \geq 0$ but the complete job sequence is not known beforehand. On the contrary, the optimal offline algorithms have clairvoyance over all released jobs in the whole input but $\kappa = 0$ for their jobs.

A metric of evaluating online algorithms

One of the widely employed metrics in evaluating online algorithms' performance is *competitive ratio* [3]. For a competitive online algorithm, we compare it with an optimal offline algorithm. The offline algorithm is a clairvoyant one, empowered to know the whole input sequence in advance to make its decision. Competitive online algorithms guarantee the worst-case performance, without requiring any stochastic assumptions on the input sequences.

Definition Competitive ratio [3]. Given a maximization problem (respectively, a minimization problem), a deterministic online algorithm ON is called *k-competitive* if its objective value on *any* instance is at least $1/k$ (respectively, at most k) of the objective value of an optimal offline algorithm on this instance: $k = \max_{\mathscr{I}}\{(\text{OPT}(\mathscr{I}) - \varepsilon)/\text{ON}(\mathscr{I}),\ \text{ON}(\mathscr{I})/(\text{OPT}(\mathscr{I}) - \varepsilon)\}$, where ε is a constant, $\text{OPT}(\mathscr{I})$ is the optimal solution of an input $\mathscr{I}$ and ON is a deterministic algorithm. The parameter k is known as the online algorithm's *competitive ratio* [3]. If the additive constant $\varepsilon \leq 0$, the algorithm ON is called *strictly k-competitive*.

Note that for online algorithms, no stochastic assumption is made on the input. The optimal offline algorithm is also called the *adversary* of the online algorithm since the input sequence constructed by the offline optimal algorithm is allowed to maximize the competitive ratio k. The *upper bounds* of competitive ratio are achieved by some known online algorithms. A competitive ratio less than the *lower bound* is not reachable by *any* online algorithm. For input sequences with sufficiently large length, we can ignore ε in the above definition.

In most scenarios, competitive ratio provides a pessimistic result of the online algorithm's performance since the adversary can adaptively generate the input to beat the online algorithm. In this paper, we define a *(weak) competitive ratio* to show the performance of an online algorithm compared with a weak adversary with less resources. For example, we allow the online algorithm to use more buffer space and study the relationship between the extra resource and improvement of competitiveness. In any case, we have no assumptions on the input sequence.

3 An Algorithm

We first note that in the slack model, each job has a slack κ. Then, any released job j must be processed by time $r_j + \kappa$, and no more than $\lfloor \kappa \rfloor$ jobs can be concurrently pending in the buffer (queue) at any time. In the model, an extra buffer with size $\lfloor \kappa \rfloor$ is sufficient for use. For simplicity of presentation, we assume κ $(= \lfloor \kappa \rfloor)$ is an integer constant in our following algorithm design and analysis. Here, we consider the variant in which the queue length of the online algorithm's buffer is always limited by κ. The queue size of the adversary is 0. Our algorithm and analysis are suitable for the more general setting when the optimal offline algorithm has a buffer with size a and the online algorithm is to allowed to have a buffer with size of $a + \kappa$.

3.1 The algorithm SLACK

The algorithm's idea has the flavor of "lazy scheduling". The machine is at its `sleep` state initially. A coming job j will be buffered into the queue before it gets expired at time $r_j + \kappa$. At the time step $r_j + \kappa$ when the first job j in the queue is to expire, the machine is powered on to the `active` state and starts to process all pending jobs in the queue in a FIFO manner. If a job is chosen to be processed, it will be removed from the queue immediately and one more queue space (buffer slot) will be available for one coming job. Once the machine is at the `active` state, it processes jobs till the queue becomes empty (that is, no pending jobs). Then the machine is powered off to its `sleep` state immediately. This algorithm is described in Algorithm 1.

Algorithm 1 SLACK(κ)

```
1: Append newly arriving jobs at the end of the queue.
2: while the machine is at its active state do
3:     if there are pending jobs in the queue then
4:         Process pending jobs in a FIFO order.
5:     else
6:         Power off the machine to its sleep state.
7:     end if
8: end while
9: while the machine is at its sleep state do
10:    if the earliest-released pending job is $j$ and the
       current time is $r_j + \kappa$ then
11:        Power on the machine to its active state.
12:    end if
13: end while
```

4 Analysis of SLACK

In this section, we provide the proofs of the correctness, of its running complexity, as well as of the worst-case performance in term of competitive ratio of the algorithm SLACK.

4.1 On correctness

Theorem 1 *The algorithm SLACK completes all jobs released. It has a running complexity of $O(n)$ where n is the number of jobs released.*

Proof Directly from Algorithm 1 we know that the running complexity is linear $O(n)$ of the number of jobs since we process each released job in constant time. Now, we prove that all released jobs can be finished by the algorithm.

We apply the contradiction method to prove the correctness of the algorithm. Assume a job j is released but it is not processed by the machine by time $r_j + \kappa$. We know that if j is buffered, j cannot be removed from the buffer due to following released jobs. We case study j.

Assume j has not been buffered.

In this case, at time r_j, the buffer has κ jobs pending and the machine has not processed the earliest-released job. However, this cannot happen since in each time step, at most one job is released and the earliest job, say f, must be released at time $r_f \leq r_j - \kappa$. Thus, it is time to process that job since $r_f + \kappa \leq r_j$. This part of analysis also infers that in the algorithm, any job should be able to be buffered at its arrival.

Assume j is buffered at time r_j.

In this case, let the earliest-released job in the buffer be f when j is buffered at time r_j. Let the queue at time r_j be Q_{r_j}. Since at most one job is released in each time step, then $r_f < r_j - |Q_{r_j} \setminus \{j\}|$. If j is not processed by time $r_j + \kappa$, f should not have been processed by time $r_j + \kappa - |Q_{r_j} \setminus \{j\}|$. This contradicts $r_f + \kappa < r_j + \kappa - |Q_{r_j} \setminus \{j\}|$.

Hence, an unprocessed job j does not exist. Theorem 1 is completed. □

4.2 On competitiveness

Theorem 2 *The algorithm SLACK has a strict competitive ratio bounded by $2C/(C + \mu\kappa)$.*

When μ is set 0, from Theorem 2, we immediately have

Corollary 1 *When an online algorithm does not have a buffer ($\kappa = 0$, jobs have no slacks), the algorithm SLACK is 2-competitive.*

Corollary 1 is the result presented in [10]. Note that when μ is set 0 for online algorithms, the lower bound of competitive ratio is 2. Thus, in the slack model, increasing μ beats *any* deterministic online algorithms with $\mu = 0$.

Proof (of Theorem 2) Our proof employs a *phase-based* charging scheme. We first define non-overlapping *phases* to cover all the schedule created by the online algorithm SLACK. We then prove that in each phase, with an appropriate charging scheme, SLACK is $(2C/(C + \mu\kappa))$-competitive; this directly results that SLACK's competitive ratio is $2C/(C + \mu\kappa)$ over the whole schedule.

Let $\mathscr{I}$ be such a sequence of jobs. Assume that the earliest-released job satisfying the following conditions be j.

Condition 1 *The job j is released at time r_j with slack κ. This job j is either followed by a set of jobs $\hat{J} = \{j_1, \ldots, j_l\}$ satisfying that for job j_i, it has $r_{j_i} < r_{j_{i+1}}$ and $r_{j_i} \le r_j + \kappa + i$, where $i = 1, 2, \ldots, l$; or is followed by nothing within the time interval $[r_j, r_j + \kappa + 1]$. That is $\hat{J} = \emptyset$.*

According to the algorithm, this set of jobs $\{j\} \cup \hat{J}$ ($= \{j, j_1, \ldots, j_l\}$) will be processed one by one and the machine does not need to be spinning any time in between $r_j + \kappa$ and $r_j + \kappa + l$. Let the next job satisfying Condition 1 but not in $\{j, j_1, \ldots, j_l\}$ be j' (if any) and S be the last time step SLACK processes a job. The duration $[r_j + \kappa, \min(r_{j'} + \kappa - 1, S)]$ is defined as a *phase*. For the case $\hat{J} = \emptyset$, the *phase* $[r_j + \kappa, r_j + \kappa + 1]$ only includes one job j.

Note that the machine will not be spinning and the queue of the online algorithm is empty both at the beginning and the end of each phase. In each phase defined by SLACK, we compare SLACK with an optimal offline algorithm OPT which has no queue for pending jobs and processes a job immediately at its arrival. This means that SLACK and OPT process the same set of jobs during each phase.

Now, we examine that whether the online algorithm and the offline algorithm are in the same state at the end of each phase or not. We introduce two parameters S_{ON} and S_{OPT} to represent the states of the online algorithm SLACK and the offline algorithm OPT at the end of each phase. Let ALG $\in$ {ON, OPT}.

$$S_{\text{ALG}} = \begin{cases} 1, & \text{if ALG is at its } \texttt{active} \text{ state} \\ & \text{after processing the last job of the phase,} \\ 0, & \text{otherwise.} \end{cases}$$

Energy cost of the online algorithm SLACK

Assume there are n_k jobs for the k-th phase. For SLACK, the machine is powered on to an `active` state at time $r_j + \kappa$ and it keeps in the `active` state in processing jobs continuously for n_k time units. The machine gets to the `sleep` state immediately after processing the last job in this phase. The machine is powered on/off once for each phase. The total energy cost of SLACK for the k-th phase is $C_{\text{ON}}(\mathscr{I}_k) = C + \mu n_k$.

Energy cost of the optimal offline algorithm OPT

In the same phase, if the machine is at its `sleep` state, OPT powers on the machine to an `active` state when it sees the first job's arrival. Based on the length of the idling period T between this job and the next released job, the machine is either spinning (if $T < C/\mu$) or getting to the `sleep` state (if $T > C/\mu$). Denote $l_{i,i+1}$ as the idling period length between the i-th job and the $(i+1)$-th job. We define

$$C_{i,i+1} = \begin{cases} C, & \text{if } l_{i,i+1} \ge C/\mu, \\ \mu \cdot l_{i,i+1}, & \text{otherwise.} \end{cases}$$

For the same phase, the cost of OPT is represented as $C_{\text{OPT}}(\mathscr{I}_k) = C + \mu n_k + \sum_{i=1}^{n_k - 1} C_{i,i+1}$.

For SLACK, it must be at the `sleep` state at the end of each phase. For OPT, its state depends on the length of the interval between the last release time of a job in the previous phase and the first release time of a job in the following phase. We case study OPT's state at the end of a phase.

Assume OPT is at a `sleep` state at the end of the phase. OPT will never beat the online algorithm for its extra pay on spinning over all $l_{i,i+1}$. In this case, SLACK's performance is no worse than that of OPT.

Assume OPT is at an `active` state at the end of the phase. We note that the "gap" between the last time step SLACK processes a job and the earliest released job in the next phase is no more than C/μ. Thus, OPT is spinning after processing the last job of the current phase. Let this gap's length be T_k time units. The cost of OPT for the current phase is $\hat{C}_{\text{OPT}}(\mathscr{I}_k) \ge C + \mu n_k + \mu T_k$, where $T_k > \kappa + 1$. (For any two neighboring phases, the gap length $> \kappa + 1$). Also, we have $T_k \le C/\mu$. Then $\kappa + 1 < T_k \le C/\mu$.

In order to keep SLACK share the same state `active` with OPT, we charge the cost of powering up the machine for SLACK in the next phase into the amortized cost of the current phase. We have $\hat{C}_{\text{ON}}(\mathscr{I}_k) = C + \mu n_k + C$.

For both cases, we have

$$\begin{aligned} \hat{C}_{\text{ON}}(\mathscr{I}_k) &= C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot C. \\ \hat{C}_{\text{OPT}}(\mathscr{I}_k) &= C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot \mu \cdot T_k. \end{aligned}$$

Let SLACK's competitive ratio be α. We ensure $\hat{C}_{\text{ON}} \le \alpha \cdot \hat{C}_{\text{OPT}}$.

$$\alpha = \frac{\hat{C}_{\text{ON}}}{\hat{C}_{\text{OPT}}} = \frac{C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot C}{C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot \mu \cdot T_k}.$$

Remember that $T_k \ge \kappa$, so we have

$$\begin{aligned} \hat{C}_{\text{OPT}}(\mathscr{I}_\kappa) &= C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot \mu \cdot T_k \\ &\ge C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot \mu \cdot \kappa. \end{aligned}$$

$$\begin{aligned} \alpha &= \frac{C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot C}{C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot \mu \cdot T_k} \\ &\le \frac{C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot C}{C + \mu n_k + (S_{\text{OPT}} - S_{\text{ON}}) \cdot \mu \cdot \kappa} \\ &\le \frac{C + \mu n_k + C}{C + \mu n_k + \mu \cdot \kappa} \le \frac{2C}{C + \mu \cdot \kappa}. \end{aligned}$$

Note that if $\kappa \geq C/\mu$, the online algorithm will always beat the offline algorithm. Also, SLACK's competitive ratio α is getting better when the buffer size κ is increased. Above ratio α holds for each phase. Theorem 2 is proved. □ □

We remark here that Theorem 2 provides the worst-case guarantees. In practice, the algorithm's performance might be far better than the bound presented. Figure 2 shows the change of the competitive ratio along the increase of packet slacks.

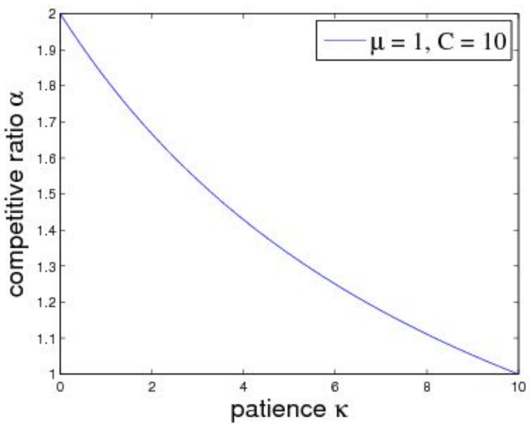

Figure 2: Competitive ratio α along the increase of packet slacks κ

4.3 The lower bound

In this section, we present the lower bound of competitive ratio β for deterministic online algorithms for the patient model. Here, we consider an *adaptive offline adversary*, who generates a job sequence based on the past behavior of the online algorithm along the course of the scheduling. We refer to the optimal offline algorithm as OPT and online algorithm as ON respectively.

Note here that from Theorem 2, if $\kappa \geq C/\mu$, we know that there exists an optimal online algorithm such as SLACK. Thus, we only consider the lower bound of competitive ratio when $\kappa < C/\mu$. The main technical contribution here is to show that even with facility of (limited) slacks for jobs, any online algorithm cannot perform arbitrarily close to the optimal offline algorithm.

Theorem 3 *Assume $\kappa < C/\mu$. The lower bound of competitive ratio for deterministic algorithms is*

$$\beta \geq \min\left(\frac{3\mu+2C}{2\mu+2C}, \frac{2\mu+2C}{2\mu+C+\mu\cdot\kappa}\right).$$

Proof Initially, we assume that both OPT and ON have their machines at the `sleep` state. Without loss of generality, we assume C/μ is an integer and μ is the energy cost per unit time when the machine is `active` (including spinning).

Let the adversary release a job j_0 at the beginning of step 1. OPT powers on the machine to process j_0 and gets it done at the end of step 1. For ON, there are two options now: (1) letting the machine transit to its `active` state to process j_0, or (2) buffering j_0 into the queue till some time $t \leq 1+\kappa$, where $1+\kappa$ is the latest time for j_0 to be executed.

Assume ON powers on the machine to the `active` state immediately at j_0's arrival.

After completing j_0 at the end of step 1, ON either keeps the machine `active` (spinning) or turns to the `sleep` state.

Assume ON keeps in the `active` state. The adversary then releases another job j_1 at time $C/\mu+2$. There are no more jobs released.

In this case, OPT powers off the machine after finishing j_0 at the end of step 1 and then powers on to an `active` state at time $C/\mu+2$ to process job j_1. The total cost paid by OPT is $C^1_{\text{OPT}} = 2\mu+2C$.

On the other hand, ON either keeps the machine `active` till j_1 arrives with a total cost of $C^1_{\text{ON}} = 3\mu+2C$, or ON makes the machine `sleep` at some time before $C/\mu+2$ and powers it on to the `active` state at some time before the deadline of job j_1 to process it with a total cost $\geq C^2_{\text{ON}} = 3\mu+2C$.

The lower bound of competitive ratio β_1 here is

$$\beta_1 \geq \frac{\min(C^1_{\text{ON}}, C^2_{\text{ON}})}{C^1_{\text{OPT}}} = \frac{3\mu+2C}{2\mu+2C} \tag{1}$$

Assume ON powers off the machine to its `sleep` state at the end of time step 1. The adversary releases a job j_2 at the beginning of step 2. There are no more jobs released.

OPT keeps the machine `active` to finish job j_2 with a total cost of $C^2_{\text{OPT}} = 2\mu+C$.

On the other hand, ON has chosen to power off the machine to `sleep`. To get j_2 finished, it will need to switch back to the `active` state again at some time $t \leq \kappa+2$, the latest time for j_2 to be executed. The total cost of ON is $C^3_{\text{ON}} \geq 2\mu+2C$.

The lower bound of competitive ratio β_2 here is

$$\beta_2 \geq \frac{C^3_{\text{ON}}}{C^2_{\text{OPT}}} = \frac{2\mu+2C}{2\mu+C}. \tag{2}$$

Assume ON buffers the job j_0 into the queue.

ON must switch the machine to `active` to process j_0 at some time $t \leq \kappa + 1$. After time $t+1$, ON will either choose to power off the machine to a `sleep` state or keep it spinning. We consider the end of step $t+1$ now.

Assume ON keeps `active`. The adversary adopts a strategy similar to the strategy used above. It releases another job j_3 at time $t + C/\mu + 2$. There are no more jobs released.

In this case, OPT powers off the machine to the `sleep` state after finishing job j_0 at time 1. Then it switches to the `active` state at time $t + C/\mu + 2$ to process job j_3. The total cost paid by OPT is $C^3_{\text{OPT}} = 2\mu + 2C$.

On the other hand, ON will either keep in the `active` state till the job j_3 finished with a total cost of $C^4_{\text{ON}} = 3\mu + 2C$, or ON switches to the `sleep` state at some time before $t + C/\mu + 2$ and powers on the machine at some time before $t + C/\mu + 2 + \kappa$ to process the job j_3. The total cost of ON is $C^5_{\text{ON}} \geq 3\mu + 2C$.

The lower bound of competitive ratio β_3 here is

$$\beta_3 \geq \frac{\min(C^4_{\text{ON}},\, C^5_{\text{ON}})}{C^3_{\text{OPT}}} = \frac{3\mu + 2C}{2\mu + 2C}. \tag{3}$$

Assume ON powers off the machine at time $t+1 \leq \kappa + 1$ after finishing the job j_0. (Note $t \leq \kappa$.) Only in this case, we need the assumption in Theorem 3 that $\kappa < C/\mu$. The adversary releases a job j_4 at time $t+1$. There are no more jobs released.

OPT keeps the machine `active` till finishing job j_4 at time $t+2$. The total cost of OPT is $C^4_{\text{OPT}} = 2\mu + C + \mu \cdot t$.

On the other hand, ON powers the machine off and it needs to turn it on again before time $t + \kappa + 1$ to process job j_4. The total cost of ON is $C^6_{\text{ON}} \geq 2\mu + 2C$.

The lower bound of competitive ratio β_4 here is

$$\beta_4 \geq \frac{C^6_{\text{ON}}}{C^4_{\text{OPT}}} = \frac{2\mu + 2C}{2\mu + C + \mu \cdot t} \geq \frac{2\mu + 2C}{2\mu + C + \mu \cdot \kappa}. \tag{4}$$

Combine Inequalities 1 2 3 and 4, we have the lower bound β of competitive ratio for the slack model.

$$\beta \geq \min(\beta_1,\, \beta_2,\, \beta_3,\, \beta_4) = \min\left(\frac{3\mu + 2C}{2\mu + 2C},\, \frac{2\mu + 2C}{2\mu + C + \mu \cdot \kappa}\right).$$

Therefore, β depends on the value of μ.

$$\beta = \begin{cases} \frac{3\mu+2C}{2\mu+2C}, & \text{when } \mu \leq \min\left(C/\mu,\, \frac{2C^2+\mu C-2\mu^2}{3\mu^2+2\mu C}\right). \\ \frac{2\mu+2C}{2\mu+C+\mu\cdot\kappa}, & \text{when } \frac{2C^2+\mu C-2\mu^2}{3\mu^2+2\mu C} \leq \kappa \leq C/\mu. \end{cases}$$

Theorem 3 is completed. □ □

5 Related Work and Conclusion

In this paper, we consider scheduling algorithms for jobs with slacks for energy management. The goal is to minimize the total cost upon giving slacks to jobs in an online setting. We focus on investigating the relationship between the competitive ratio and jobs' slacks. We prove that with appropriate slacks, an online algorithm can beat the weak optimal offline algorithm without slacks. We also present the lower bound of competitive ratio that any deterministic online algorithm cannot achieve. We provide theoretical analysis of the model and the online algorithm SLACK. These results are provable worst-case guarantees.

Two related work are [4] and [13]. In both work, buffers can used to smooth the variations of job processing time and also used for elongating idle periods. In their settings, larger buffers may not be better than multiple smaller buffers as smaller buffers cost less energy. In contrast to their approaches, we consider introducing extra buffer space and reducing energy cost in total. The energy overhead from the lightly larger buffer space is negligible in our setting.

There are some problems still open: We need to shrink the gaps between the lower bound and upper bound of competitive ratio for the slack model. We still do not know the lower bound and upper bound of competitive ratio when jobs are with variable lengths (rather than unit-lengths jobs). We also have not addressed the same problem (online minimizing energy consumption) under the fading channel models [6] where the fade state of the channel determines the throughput obtained per unit of time and the channel's quality changes over time as well. We also note here that along with stochastic assumptions on the job arrival patterns, better competitiveness can be achieved. Our future research also targets on multiple buffer management.

Acknowledgements

This research is partially supported by NSF under Grant CCF-0915681.

References

[1] J. Augustine, S. Irani, and C. Swamy. Optimal power-down strategies. *SIAM Journal on Computing (SICOMP)*, 27(5):1499–1516, 2008.

[2] P. Baptiste, M. Chrobak, and C. Durr. Polynomial time algorithms for minimum energy scheduling. In *Proceedings of the 15th Annual European Symposium on Algorithms (ESA)*, pages 136–150, 2007.

[3] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[4] L. Cai and Y-H Lu. Energy management using buffer memory for streaming data. *IEEE Transactions on Computer Aided Design of Integrated Circuits System (IEEE TCAD)*, 24(2):141–152, 2005.

[5] G. Dhiman and T. S. Rosing. System-level power management using online learning. *IEEE Transactions on Computer-Aided Design of Integraded Circuits and Systems (IEEE TCAD)*, 28(5):676–689, 2009.

[6] A. Fu, E. Modiano, and J. Tsitsiklis. Optimal transmission scheduling over a fading channel with energy and deadline constraints. *IEEE Transactions on Wireless Communications*, 6(1):630–641, 2006.

[7] S. Irani and K. R. Pruhs. Algorithmic problems in power management. *ACM SIGACT News*, 36(2):63–76, 2005.

[8] S. Irani, S. Shukla, and R. Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(3):325–346, 2003.

[9] S. Irani, G. Singh, S. K. Shukla, and R. K. Gupta. An overview of the competitive and adversarial approaches to designing dynamic power management strategies. *IEEE Transactions on Very Large Scale Integration Systems (IEEE TVLSI)*, 13(2):1349–1361, 2005.

[10] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11:542–571, 1994.

[11] P. Krishnan, P. Long, and J. Vitter. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. *Algorithmica*, 23(1):31–56, 1999.

[12] A. Maxiaguine, S. Kunzli, S. Chakraborty, and L. Thiele. Rate analysis for streaming applications with on-chip buffer constraints. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference (Asia-DAC)*, pages 131–136, 2004.

[13] J. Ridenour, J. Hu, N. Pettis, and Y-H Lu. Low-power buffer management for streaming data. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(2):143–157, 2007.

[14] 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 (FOCS)*, pages 374–382, 1995.