Energy-Efficient Online Scheduling with Deadlines

Aaron Coté, Adam Meyerson, Alan Roytman, Michael Shindler, Brian Tagiku
{acote,awm,alanr,shindler,btagiku}@cs.ucla.edu

Department of Computer Science, University of California, Los Angeles

Technical Report UCLA-CSD-100029

September 24, 2010

Abstract

Whether viewed as an environmental, financial, or convenience concern, efficient management of power resources is an important problem. In this paper, we explore the problem of scheduling tasks on a single variable-speed processor. Our work differs from previous results in two major ways. First, we consider a model where not all tasks need to be completed, and where the goal is to maximize the difference between the benefit of completed tasks and the cost of energy (previous work assumed that all tasks must be completed). Second, we permit a wide range of functions relating task completion time to energy (previous work assumed a polynomial relationship).

We begin by exploring multiple speed packet scheduling, and we develop 2-competitive algorithm where tasks are unit-sized and indivisible. This extends to a fractional version where benefit can be obtained for partially-completed tasks, and also extends to permit arbitrary non-negative relationships between task value and completion time. The proof introduces a novel version of online maximum-weight matching which may be of independent interest.

We then consider the problem of processor scheduling with preemption. We develop a randomized poly-logarithmic competitive algorithm by showing how to effectively "guess" a speed close to that which the optimal solution will use. We also prove a number of lower bounds, indicating that our result cannot be significantly improved and that no deterministic algorithm can be better than polynomially-competitive.

We also consider the case where all tasks must be completed by their deadlines and the goal is to minimize energy, improving upon the best previous competitive result (as well as extending to arbitrary convex functions). Finally, we consider a problem variant where speedup affects distinct tasks differently, and provide a logarithmic-speedup competitive result and matching lower bounds.

1 Introduction

Energy-efficiency has become an increasingly important issue in computer science research [1]. This is certainly motivated in part by the alarming link between energy usage and climate change [29] and the fact that large data centers require billions of dollars worth of energy to maintain [31]. As a result, energy-saving techniques offer significant environmental and monetary incentives. Yet, such methods also offer practical benefits in the much smaller embedded platform scale where battery life is a critical factor to the viability of the system [32].

In this paper, we describe online competitive algorithms and lower bounds for a variety of scheduling problems motivated by energy conservation. Each problem concerns scheduling tasks on a single variable-speed processor. Faster processor speeds require greater amounts of energy per unit time, so we wish to throttle the processor speed so as to minimize our total energy usage. It is easy to show difficulties in completing all tasks by deadlines in an energy-efficient manner, thus we consider instead optimizing throughput, maximizing the total number or value of tasks completed. In conjunction with task deadlines, this allows a system to be responsive (in particular, it finishes a task or reports that it cannot perform a task quickly) while maintaining energy-efficiency. However, if all tasks must be finished by their deadlines, we also show how resource augmentation can help us be competitive.

The major differences between our work and prior results (*e.g.* [2, 4, 5, 6, 9, 23, 33]) are two-fold: First, our model assumes that tasks have hard deadlines, whereas most of the previous papers (all but [33]) use a weighted flow time model for quality of service. In addition to being able to accomodate time-sensitive tasks, the deadline model has the advantage of prioritizing the completion of tasks which have been in the system for a long time, inducing a measure of fairness; in comparison, weighted flow time may starve some tasks indefinitely while completing other tasks of comparable weight. Our results are the first to consider variable-speed scheduling of tasks with deadlines to maximize throughput ([33] requires all tasks be completed by their deadlines). Second, our model permits arbitrary relationships between speed and energy whereas previous work assumes a smooth polynomial relationship (except [5] which needed additional assumptions, either that all tasks had the same weight, or that rewards could be obtained for completing fractions of tasks). Real processors typically run only at a discrete set of possible speeds [10] and even then, task completion rate need not depend linearly on processor speed; for example the running time for a very memory intensive task will not change much when the processor speed is doubled, whereas a purely computational task will take half the time at twice the processor speed. Our work is the first to allow arbitrary relationships between speed and energy in a general task model and also the first to combine arbitrary relationships between speed and energy with the deadline model ([5] uses weighted flow time rather than deadlines; [33] assumes a smooth polynomial relationship between speed and energy).

Our first problem is a packet scheduling variant with multiple speeds (packet scheduling is a particularly well-used model in wireless networks). Packets arrive online and each have a deadline and a value. Time passes in discrete timesteps and we can send greater numbers of packets each timestep for increasingly larger energy costs. Our goal is to maximize our total profit, *i.e.* total value of packets we send on time minus the total energy cost. We design a 2-competitive deterministic online algorithm for this problem via a simple greedy technique. This can be viewed as a task scheduling result; however, there are a number of inherent assumptions (packets are atomic, have identical "workload," and complete quickly relative to the rate of arrivals even at a slow speed); these assumptions are equivalent to a fractional model where we can obtain benefit from completing part of a task (such a model is considered, for example, in [5], albeit under a different quality of service measure). Interestingly, our algorithm works for a novel version of online maximum weighted bipartite matching, where nodes on one side of the matching arrive online along with their incident edges. While there are strong lower bounds for this problem if matches made are permanent [27, 3, 13, 16, 20, 28], we instead assume that the nodes on the other side of the matching (not the online-arriving nodes) lock in an online manner. Only when a node locks is its match fixed permanently. In fact, this allows us to assume the value of a packet depends upon the completion time in an arbitrary way (for example packets which decay exponentially in value over time).

We then move on to scheduling tasks with preemption. Here, tasks have arrival times, deadlines, workloads, and values. We can run at faster speeds for larger energy costs, and our goal is to maximize the total value of tasks completed by their deadlines minus the cost of energy. Note that the matching approach will not directly work since we might try to match only part of a task. For this problem, we provide a randomized poly-logarithmic competitive algorithm for this problem. We do this by showing how to "guess" a speed to run each task in a manner that is not much worse than optimum. This reduces the problem to the single-speed case, where we can divide the tasks into a poly-logarithmic number of buckets, randomly select one of these buckets and run the algorithm of [8]. We also present some new logarithmic lower bounds which suggest that, even allowing preemption, we cannot do much better than our result in the variable-speed case. We also show that any deterministic algorithm must have a polynomial competitive ratio.

Our third problem does not directly involve energy management, but will be useful later. We consider the version of packet scheduling (at a single speed) where the goal is to minimize the total value of packets which are dropped. We prove that unlike the maximization version, no deterministic competitive algorithm is possible for this problem unless the online algorithm is allowed to transmit more packets per time step than the offline. Further, this "speedup factor" must be super-constant. We show that speedup factors logarithmic in various natural parameters are sufficient. Perhaps surprisingly, if the packets have agreeable deadlines (earlier arrival implies no later deadline) then constant speedup suffices. This is the first provable gap in competitive ratio between agreeable deadlines and the general case.

Finally, we consider preemptive scheduling of tasks where all tasks must be completed by their deadlines. We show that provided our algorithm can run a constant times faster than optimum for the same cost in energy, we can obtain optimality; it is also easy to prove that such speedup is necessary. We show that in the special case where energy is a polynomial function of speed, our approach improves upon the previous result of [33] (in addition our result permits arbitrary convex relationships between speed and energy, and has a much simpler proof). Finally, we consider a model where the effect of increasing processor speed is different for distinct tasks. We give a competitive-ratio-preserving reduction from this problem to packet scheduling where we must minimize the value of dropped packets. It follows that super-constant speedup factors will be necessary (barring agreeable deadlines), and that speedup logarithmic in natural parameters is sufficient.

1.1 Related Work

In standard packet scheduling, packets arrive in an online fashion. Each packet p is specified by its arrival time a_p, deadline d_p and value v_p. At each time step t, we are to choose at most one packet p to send such that $a_p \leq t \leq d_p$. Our goal is to maximize the total value of packets sent. The online packet scheduling problem first appeared in 2001 in [19] and a deterministic lower bound of the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$ was shown in [15] shortly after. The simple greedy algorithm (sending highest value packet first) is known to be 2-competitive [19]. Since then, a number of slight improvements have been made in [12, 26, 14] with the current best being a $2\sqrt{2} - 1 \approx 1.828$-competitive algorithm. Interestingly, a ϕ-competitive algorithm is known when packet deadlines are agreeable ($a_p < a_q$ implies $d_p \leq d_q$) [25]. For randomized algorithms, a lower bound of 1.25

and an algorithm achieving competitive ratio $\frac{e}{e-1}$, where e is the base of the natural log, are given in [7]. However, to the best of our knowledge we are the first to consider a variable-speed variant of packet scheduling.

For brevity, we will not discuss in great detail the vast history of task scheduling and instead point the reader to excellent surveys in [30]. Much previous work exists in the single-speed, single-processor case such as in [8, 11, 17, 18, 21, 24], and also multi-processor variants [8, 22]. Even with only a single speed available, no algorithm can obtain better than a logarithmic competitive ratio on the total value obtained [22], although constant is possible if tasks are unweighted [8] (in expectation) or with resource augmentation [11, 17, 21, 24]. Again, our model differs from these by allowing the processor to run at various speeds. One approach is to divide tasks into tiny packets and use matching; however, this might try to complete only part of a task (thus spending large amounts in energy for no benefit). Instead, we observe that the optimum can be assumed to run each task at only a single speed (otherwise we can obtain better energy with the same completion times by averaging out the speed, using convexity).

2 Multi-Speed Packet Scheduling to Optimize Benefit

We are given a single processor with multiple speed settings and a set of packets to process. A function $c : \mathbb{Z}^+ \to \mathbb{R}$ is our energy cost to send s packets per timestep. We assume throughout this work that c is a convex, increasing function. Packets arrive in an online fashion, and each packet p is described by the tuple (a_p, d_p, v_p): the arrival time, deadline, and value of the packet (respectively). Our job is to schedule packets in order to maximize the total value of packets transmitted by their deadlines, minus the total energy cost. We observe that this can be viewed as matching packets to "slots" where slot (t, i) represents the i^{th} packet transmitted at time t. The goal is then to produce a maximum-weight matching, where a matching edge from p to (t, i) exists if $a_p \leq t \leq d_p$ and has weight $w_{p,(t,i)} = v_p - [c(i) - c(i-1)]$. The convexity of $c(s)$ implies that slots with higher i-value have lower weight edges and thus a maximum-weight matching will use the first i slots at time t (for some i), obtaining a total value equal to the sum of values of transmitted packets minus $c(i)$. We can thus model our problem as a variant of online maximum-weight bipartite matching, the formal description of which follows.

Problem 1 (Online Maximum-Weight Matching with Vertex Locking)**.** *A set of nodes B is given. Nodes A arrive online, each node $a \in A$ arriving along with its weighted incident edges to B. We must construct a matching of A to B of (approximately) maximum weight. However, when we decide to match $a \in A$ to $b \in B$, this decision is not necessarily fixed for all time. Instead, the nodes of B lock one by one in an online manner. When a node $b \in B$ locks, it must keep its current match (or unmatched status) from A forever. Note that we are not required to match all nodes of A or B, although unmatched nodes contribute zero to the objective.*

For our packet scheduling problem, B corresponds to the slots and A to the packets. A particular slot (t, i) locks when time t passes and we actually transmit packets. Any competitive algorithm for the matching problem will give the same competitive ratio for packet scheduling. In addition, we can permit more complicated weights on the matching edges, allowing us to consider cases where the (non-negative) value of a packet diminishes over time. Also note that we can approach problems where the tasks *are not* unit length by dividing the tasks into small pieces (packets), provided that completing a fraction μ of the work of task i results in μv_i value. A similar model is used in [5].

The main result of this section is a 2-competitive deterministic algorithm for this problem. To simplify the analysis, we assume that at each time step either one new node of A arrives or one new node of B locks. We designate A_t as the set of nodes from A which have arrived by time t.

Our algorithm proceeds as follows. Let F_t be the set of matching edges selected for locked nodes. At each time, we compute $\mu(A_t, B, F_t)$, the maximum-weight matching of A_t with B with the requirement that the edges F_t be included in the matching (F_t can include null edges for unmatched locked nodes). Whenever a node locks, we add the appropriate edge from our current matching to F_t. Let $f(A_t, B, F_t)$ be the value of matching $\mu(A_t, B, F_t)$.

Suppose that node $a \in A$ arrives at time t. Define $\Delta_a = f(A_t, B, F_t) - f(A_{t-1}, B, F_{t-1})$, the change in the value of the optimum matching due to this arrival. For any $b \in B$, define $\rho_t(b) = f(A_t, B, F_t) - f(A_t, B - \{b\}, F_t)$. For any $b \in B$, let ν_b be the weight of the matching edge for b in the algorithm's final solution (or $\nu_b = 0$ if b is unmatched). We observe that the algorithm's total weight is given by $ALG = \sum_{b \in B} \nu_b = \sum_{a \in A} \Delta_a$

Lemma 1. *The value of $\rho_t(b)$ is non-decreasing with time.*

Proof. If at time t a new vertex becomes locked then $F_{t-1} \subset F_t$ and $A_{t-1} = A_t$. Since we pick the appropriate edge from $\mu(A_t, B, F_{t-1})$ we have $f(A_t, B, F_t) = f(A_t, B, F_{t-1})$. This new locked edge can only reduce the weight of the best matching to $B - \{b\}$ so $f(A_t, B - \{b\}, F_t) \leq f(A_t, B - \{b\}, F_{t-1})$. Thus, $\rho_t(b) \geq \rho_{t-1}(b)$.

If at time t a new vertex a arrives, then $A_{t-1} \subset A_t$. One way to calculate $f(A_t, B - \{b\}, F_t)$ is by network flow. Add a source vertex incident to each vertex in A_t and a sink vertex adjacent to all vertices in $B - \{b\}$, each with weight 0 and capacity one. All edges between A_t and $B - \{b\}$ will remain unchanged and have capacity one. We can find a maximum-weight, maximum flow to identify our maximum-weight matching (since all capacities are integral, we know our flow precisely defines a matching). We augment B with "dummy" vertices and connect them to the vertices of A with edges of capacity one and weight zero, so as to guarantee that the max-weight flow will saturate all edges from source to A_t. We can also do this to calculate $f(A_{t-1}, B, F_t)$.

Consider taking the two graphs from these two instances and superimposing them (by merging identical vertices). Capacities of any identical edges will add, but we keep their weights constant. Now, all edges are capacity 2 except for those edges incident on a or b. Notice that the individual flows for these two graphs simply add and still form a valid flow. If we consider a maximum-weight flow f^* over this new graph, then clearly $weight(f^*) \geq f(A_t, B - \{b\}, F_t) + f(A_{t-1}, B, F_t)$.

We now show that f^* can be decomposed into flows for $f(A_t, B, F_t)$ and $f(A_{t-1}, B - \{b\}, F_t)$ which proves our claim. It is clear that in f^* the unweighted flow value is $2|A_t| - 1$. Thus, each A_t vertex except a has two units of flow passing through it. So starting from a, we can follow the flow, alternating between B and A vertices. This process must stop at a B vertex and so the path has an odd number of edges. We can then add the flow along each odd edge to $f(A_t, B, F_t)$ and the flow along each even edge to $f(A_{t-1}, B - \{b\}, F_t)$. Once this is done, we remove this flow. Note that A vertices now either have 0 units of flow, or 2 units of flow. If b was not involved in this path, then we can start from b and do the same process (being sure to add the flow involving b to $f(A_t, B, F_t)$). Again, the A vertices still have either 0 or 2 units of flow. Once a and b are handled, then the remainder of the flow can similarly be decomposed. This constructs valid flows for $f(A_t, B, F_t)$ and $f(A_{t-1}, B - \{b\}, F_t)$. Thus $weight(f^*) \leq f(A_t, B, F_t) + f(A_{t-1}, B - \{b\}, F_t)$. This gives us:

$$f(A_t, B - \{b\}, F_t) + f(A_{t-1}, B, F_t) \leq f(A_t, B, F_t) + f(A_{t-1}, B - \{b\}, F_t)$$
$$f(A_{t-1}, B, F_t) - f(A_{t-1}, B - \{b\}, F_t) \leq f(A_t, B, F_t) - f(A_t, B - \{b\}, F_t)$$
$$\rho_{t-1}(b) \leq \rho_t(b)$$

□

Theorem 1. *The algorithm is 2-competitive.*

Proof. Consider any nodes $a \in A$ and $b \in B$. Suppose that a arrives at time t, and that ab is a possible match (this requires that b is not yet locked at time t). We can write the following:

$$\begin{aligned}\Delta_a + \rho_{t-1}(b) &= [f(A_t, B, F_t) - f(A_{t-1}, B, F_t)] + [f(A_{t-1}, B, F_t) - f(A_{t-1}, B - \{b\}, F_t)] \\ &= f(A_t, B, F_t) - f(A_{t-1}, B - \{b\}, F_t).\end{aligned}$$

The last expression must be at least w_{ab} since one way to form the matching $\mu(A_t, B, F_t)$ involves taking the matching $\mu(A_{t-1}, B - \{b\}, F_t)$ and augmenting by the edge (a, b). If we let τ be the final time for the algorithm (at which we can assume all of B is locked) then by lemma 1 we have $\Delta_a + \rho_\tau(b) \geq w_{ab}$. By definition, we have $\nu_b = \rho_\tau(b)$. Summing both sides over pairs (a, b) which are matched in the optimum offline solution completes the proof. □

3 Multi-Speed Task Scheduling to Optimize Benefit

Problem 2 (Benefit-Optimized Scheduling)**.** *A set of n tasks (ℓ_i, a_i, d_i, v_i) arrives in an online fashion. Here ℓ_i is the workload, a_i is the arrival time, d_i is the deadline, and v_i is the value of the task. We are given a single processor which can run at variable speeds s, along with a convex function $\hat{c}(s)$ relating speed to energy per unit work. We must find a task and speed schedule to maximize the total value of tasks performed by their deadlines minus the total energy.*

This extends our packet scheduling model from section 2 to variable length tasks. Note that $\hat{c}$ measures cost per unit work; requiring this to be convex is a stronger condition than requiring cost per unit time to be convex. Prior work restricts to $\hat{c}(s) = s^{p-1}$ for some $p \geq 2$, which satisfies the condition. The examples justifying arbitrary relationships between speed and energy normally imply a faster-growing function. Our algorithm will work even if $\hat{c}(s)$ differs for distinct tasks. We also note that if we are restricted to a discrete set of speeds, we can interpolate between them (allowing the processor to switch rapidly between two consecutive discrete speeds) to produce a continuous result.

A direct matching approach will not work, as we might attempt to complete only a fraction of a task. In fact, even when there is only a single speed there are logarithmic lower bounds on the online competitive ratio [18]; we provide additional lower bounds in Appendix A. We will obtain a poly-logarithmic result by assigning a speed to each task, randomly selecting a set of similar tasks, and then executing the algorithm of [8]. We first explore some properties of optimum.

Lemma 2. *There exists an optimum offline schedule using a single speed s_i^* for each task i.*

Proof. The optimum designates a single task to work on at each time. Consider all times at which the optimum works on task i. If optimum runs at different speeds at these times, then an appropriate weighted average of the speeds will perform the same amount work in the same total time while reducing the cost. □

Given the optimum speed s_i^* for each task, we can reduce this to the single-speed problem where each task now has new length $\ell_i' = \ell_i / s_i^*$ and value $v_i' = v_i - \ell_i \hat{c}(s_i^*)$. Unfortunately, these speeds are not provided to us *a priori*, so we must choose appropriate speeds ourselves.

If we wish to do task i, the slowest speed we can run at is $s_i^{\min} = \ell_i/(d_i - a_i)$. We would never run i at a speed that is not profitable, so the fastest speed we would use is $s_i^{\max}$ where $v_i - \ell_i \hat{c}(s_i^{\max}) = 0$. Thus, we wish to select a speed s_i that satisfies $s_i^{\min} \leq s_i \leq s_i^{\max}$. Consider running at speed s_i

where $v_i - \ell_i \hat{c}(s_i) = \frac{1}{2}\left(v_i - \ell_i \hat{c}(s_i^{\min})\right)$. That is, our profit from performing at speed s_i is half the profit we get by performing i at $s_i^{\min}$. By convexity, $\ell_i \hat{c}(2s_i) \geq 2\ell_i \hat{c}(s_i) \geq v_i + \ell_i \hat{c}(s_i^{\min})$. We conclude that $v_i - \ell_i \hat{c}(2s_i) \leq 0$ and that $s_i^{\max} \leq 2s_i$. If it happens that there is a maximum possible speed which is slower than s_i, we let s_i equal this maximum speed instead.

It is immediate from the definition of s_i that if the optimum runs some task at speed slower than s_i, it will lose at most half its benefit by accelerating that task to s_i. Thus there exists a schedule OPT' which runs each task i at speed at least s_i and obtains at least half the optimum profit. It remains to deal with the possibility that OPT' runs tasks at speeds faster than s_i.

Lemma 3. *Suppose for some D all tasks i satisfy $D/2 \leq \ell_i/s_i \leq D$. Let OPT'' be the optimum schedule subject to the restriction that whenever i is run it must be run at precisely speed s_i. Then profit(OPT'') $\geq \frac{1}{13}\cdot$ profit(OPT').*

Proof. Let S^* be the set of tasks that are completed by OPT'. For each task i, let W_i denote the time interval beginning at the time i was started by OPT' up to the time i was finished. If two tasks $j, k \in S^*$ have $W_j - W_k \neq \emptyset$ and $W_k - W_j \neq \emptyset$, then we can swap the times OPT' works on j and k so that W_j and W_k are disjoint. In particular, we can assume without loss of generality that for every $j, k \in S^*$, either W_j and W_k are disjoint, or one is strictly contained in the other.

Define a digraph G with vertex set S^*. We add a directed edge (j, k) when $W_k \subset W_j$, and there is no i where $W_k \subset W_i \subset W_j$. G is an arborescence forest. Let h be the height of G and let S_i be the vertices at height i. Thus, S_0 contains all the tasks j where W_j contains no other W_k. Moreover, for $i > 0$ task $j \in S_i$ if the largest height amongst its children is $i-1$. Note that the S_i sets partition S^* and that for any $j, k \in S_i$ we have W_j and W_k disjoint.

Consider S_i for $i \geq 2$. Each task $j \in S_i$ has at least two descendants $k_1 \in S_{i-1}$ and $k_2 \in S_{i-2}$ in G. OPT' runs each of k_1, k_2 at speed at most $2s_{k_1}$ and $2s_{k_2}$ (respectively) so each has an execution time of at least $D/4$. If we slow j's speed to s_j, then this can increase the execution time of j by at most $D/2$. So by removing k_1, k_2, we maintain a feasible schedule while running $j \in S_i$ at s_j. Thus, we obtain the profit of S_i while losing the profit of S_{i-1} and S_{i-2}.

For $i \in \{0, 1, 2\}$, let $X_i = \bigcup_{n \geq 1} S_{3n+i-1}$. If we choose the X_i with largest total profit and remove all other S_i as above, we can construct a feasible schedule with total profit at least $\frac{1}{3}$ of the total profit due to all S_i with $i \geq 2$.

Now consider only the tasks in S_i for $i = 0, 1$. Each task j is running at speed at most $s_j^{\max} \leq 2s_j$, thus j has execution time between $D/4$ and D. If we reduce the speed of each j to s_j, this at most increases the execution time of each by $D/2$. By our definition of s_j, this execution time must fit within $[a_j, d_j]$, but can overlap with any task within $D/2$ of either side of the original W_j. Extending task j's duration causes it to overlap with at most two other tasks in each direction.

We split S_i into 5 feasible sets $X_1, \ldots, X_5$ as follows. We first sort the tasks in order of W_j. Set X_i consists of the i-th task and every fifth task thereafter in the ordering. One of these sets must obtain at least $\frac{1}{5}$ the total profit of S_i.

Thus, to construct OPT'' we do the following: If S_0 contains profit at least $\frac{5}{13}\cdot \text{profit}(OPT')$, then we schedule only S_0 as described above to get total profit $\frac{1}{13}\cdot \text{profit}(OPT')$. Otherwise, we try the same with S_1. Otherwise, it must be that the profit in levels S_i for $i \geq 2$ is at least $\frac{3}{13}\cdot \text{profit}(OPT')$. Thus, we can schedule one of the X_i as above to obtain at least $\frac{1}{13}\cdot \text{profit}(OPT')$.

In all cases, we find a schedule that obeys speeds s_j and gets profit at least $\frac{1}{13}$ that of OPT'. □

Theorem 2. *There exists a randomized $O(\log V \log VL)$-competitive online algorithm where V is the ratio of maximum to minimum task value and L is the ratio of maximum to minimum length.*

Proof. For each task i we compute s_i. Note that in order to be profitable, we should never run at a speed that costs more than the maximum task value per unit work. Moreover, we only consider speeds that obtain at most half the optimum profit. Since optimum can get at most v_i from each task, the slowest speed we consider costs at least $\frac{v_{\min}}{2L}$. By the convexity of the cost function, we can conclude the ratio of the maximum to minimum speeds we consider is at most $2VL$.

Then we partition tasks into buckets by their value of $v_i - \ell_i \hat{c}(s_i)$ and by their value of ℓ_i/s_i, such that any two tasks in the same bucket are within a factor of two in each value. Note that this creates at most $O(\log V \log VL)$ buckets. We now select a bucket uniformly at random, and run the algorithm of [8] on the tasks from this bucket. Their algorithm is 2-competitive for unit values and equal execution times, and it is simple to extend this to our case for a single bucket with an increase in the constant. Also note that their algorithm is non-preemptive (even though it competes against a possibly preemptive offline optimum). In expectation our random choice of bucket costs us at most $O(\log V \log VL)$, completing the proof. □

We note that if we do not know V and L *a priori*, we can randomly select buckets in a different way. For any $\varepsilon > 0$, we can select a value i such that $\Pr[n/2 \le i \le n] > \frac{\Theta(\varepsilon)}{\log^{1+\varepsilon} n}$. This only slightly increases our competitive ratio to $O(\frac{1}{\varepsilon^2} \log^{1+\varepsilon} V \log^{1+\varepsilon} VL)$.

4 Packet Scheduling to Minimize Drop Value

We consider online packet scheduling at a single speed to minimize the total value of dropped packets. Note that this is quite similar to the problem of [19] and that the optimum offline is identical; however from a competitive perspective the change in objective makes a substantial difference. This will be used in a result for multi-speed scheduling in the next section.

Problem 3 (Packet Scheduling: PS). *A set of n packets (a_p, d_p, b_p) arrives in an online fashion. We are allowed to transmit one packet at each time step, and all packets must be transmitted between their arrival time a_p and deadline d_p (or not transmitted at all). If S is the set of packets transmitted and $N - S$ is the set of packets not transmitted, then our goal is to minimize $\Sigma_{p \in N-S} b_p$.*

We will prove that no algorithm can be competitive for this problem unless we permit a super-constant speedup (online is allowed to send more-than-constant number of packets per time). We will then provide competitive algorithms with speedup logarithmic in natural parameters.

4.1 Lower Bound

Our lower bound resembles [11], which shows that optimum maximization performance cannot be attained without super-constant speedup in a problem where tasks have variable workloads. In our problem the goal is only constant-competitiveness but on the *minimization* objective, and all packets have workload one. We will construct an example, where L and α are arbitrarily large constants and k is the speedup factor we are allowed over optimum. Our set of packets is enumerated as follows:

- $\forall j$ s.t. $1 \le j \le L$: $A_j = (1, j, \alpha^{j-1})$
- $\forall j$ s.t. $1 \le j \le L$: $B_j = (j, L + 2^{k-1}, \alpha^L)$
- $\forall j$ s.t. $1 \le j \le 2^{k-1}$: $C_j = (L+1, L + 2^{k-1}, \alpha^L)$
- $\forall i$ s.t. $0 \le i \le k-2$ and $\forall j$ st $2^i + 1 \le j \le L - 2^i + 1$: $D^i_j = (j, j + 2^i - 1, \alpha^{j+2^i-2})$

Lemma 4. *For all times t: $2^{k-1} \le t \le L$, if no more packets arrived after t, then OPT would send all packets of value $v \ge \alpha^{t-1}$.*

Proof. It suffices to show that there exists an algorithm that would be able to send all packets of value $v \geq \alpha^{t-1}$. Note that no packet of appropriate value has a deadline less than t.

- All A_j packets ($j \geq t$) must be sent. Send A_t through $A_{t+2^{k-2}-1}$ in the first 2^{k-2} timesteps.
- Send all packets $A_{t+2^{k-2}}$ and greater at their deadline.
- Send packets B_j for $1 \leq j \leq t - 2^{k-1}$ at time $j + 2^{k-2}$.
- Send packets $B_{t-2^{k-1}+1}$ through B_t during time intervals $L+1$ through $L + 2^{k-1}$.

This leaves time intervals $t - 2^{k-2} + 1$ through $t + 2^{k-2} - 1$ to send the D packets. We will prove this can be done by induction on k.

If $k = 2$, then the only D packet of value $v \geq \alpha^{t-1}$ is D_t^0, which can be sent at time t. Assume this is also true for $k = k'$. If $k = k' + 1$, then by the inductive hypothesis, we have time intervals $t - 2^{k-2} + 1$ through $t - 2^{k-3}$ and $t + 2^{k-3}$ through $t + 2^{k-2} - 1$ to send all D^{k-2} packets of value $v \geq \alpha^{t-1}$. The appropriate value packets are D_j^{k-2} $\forall j : t - 2^{k-2} + 1 \leq j \leq t$. If $j \leq t - 2^{k-3}$, send D_j^{k-2} at time j. Otherwise, send it at time $j + 2^{k-2} - 1$. □

Lemma 5. *Any algorithm ALG which obtains a finite competitive ratio on drop value can send no more than $k \cdot 2^{k-1}$ B packets by time L.*

Proof. Since α is arbitrarily large, if OPT sends all packets of value $v \geq \alpha^{t-1}$ for any given t, then ALG must do so also. Note that $\forall t$: $2^{k-1} \leq t \leq L$, exactly k packets of value α^{t-1} are due at time t: packets A_t and $D^i_{t-2^i+1}, 0 \leq i \leq k-2$. By Lemma 4, these packets must be sent by ALG. Thus, if we have a k speedup over OPT, then ALG must send less than $k \cdot 2^{k-1}$ B packets by time L. □

Theorem 3. *If we are limited to a constant speedup factor of k over OPT, there is no algorithm which can bound the competitive ratio on drop value.*

Proof. ALG may only send $k \cdot 2^{k-1}$ B and C packets after time L. OPT may send all B and C packets, there are $L + 2^{k-1}$ B and C packets, and L may be arbitrarily large. Therefore by lemma 5, ALG cannot send all B and C packets. Thus if we are limited to constant k speedup, no algorithm can obtain a finite competitive ratio on drop value. □

In fact, we can extend this further to make a claim of any speedup k, which may or may not be - and will turn out not to be - constant. Note that $k \cdot 2^k \geq L + 2^{k-1}$, and thus $(2k-1)2^{k-1} \geq L$. Solving for k, we find that if the speedup k is a function of L, it must be $\Omega(\frac{\log L}{\log \log L})$. Setting the window size to be this value L, as is a valid input, gives us the matching bound if the speedup must be a function of the window size. This allows us to state:

Theorem 4. *If any algorithm that solves this problem is to have a bounded competitive ratio, and the speedup must be a function of the window size W, then the speedup must be $\Omega(\frac{\log W}{\log \log W})$.*

4.2 Algorithmic Results

By Theorem 3, our speedup must depend on parameters of the instance. We propose two such speedups. Let W denote the maximum packet window $\max_p(d_p - a_p)$. Let V denote the ratio of the maximum to minimum packet value. We will design algorithms with speedup $O(\log W)$ and $O(\log V)$ which obtain optimum performance. Our algorithms are based on splitting up tasks into buckets, then using the speedup to simultaneously solve all buckets optimally. The main result of this section is the following theorem, the proof of which is deferred to appendix B.

Theorem 5. *Packet Scheduling admits an $O(\log W)$-speedup optimal algorithm and an $O(\log V)$-speedup optimal algorithm. If the packets have agreeable deadlines then there is an $O(1)$-speedup optimal algorithm.*

5 Multi-Speed Task Scheduling to Minimize Energy

We consider a task scheduling problem on a single processor with variable speed, where all tasks must be scheduled by their deadlines. We are allowed to suspend tasks and resume them later, implying that we can execute tasks in order of earliest deadline and the problem reduces to selecting the speed at each time step. Formally, our problem is defined as follows.

Problem 4 (Task Scheduling). *A set of n tasks (ℓ_i, a_i, d_i) arrives in an online fashion. We are given a single processor with m speeds s_j (work units/time units) which consume energy p_j (energy units/time units). Find a task and speed schedule such that ℓ_i units of work are performed on each task i between its arrival time a_i and deadline d_i, and the total energy consumption is minimized.*

We cannot be competitive for this problem without speedup: if there is a maximum speed and the algorithm does not use it, later arriving tasks can force the algorithm to violate deadlines. If the algorithm always runs at the maximum speed, the energy will not be competitive.

We order the speeds such that $s_{j+1} > s_j$ and assume that for any j we have $\frac{p_{j+1}}{s_{j+1}} \geq \frac{p_j}{s_j}$ and $s_{j+1} \geq \alpha s_j$. If this fails to hold, we can eliminate energy-inefficient speeds for no cost (the first inequality) and we can eliminate too-similar speeds from consideration for a speedup factor of α (the second inequality). We will also assume that there is a speed zero with $s_0 = 0$ and $p_0 = 0$ (this assumption can be easily removed, but makes our proofs simpler). Our algorithm will run a number of virtual machines in parallel. The j^{th} virtual machine M_j will always run at speed s_j if it has any work to do, or at speed zero otherwise. When a task arrives, we will schedule as much work as possible on machine one, then place as much of the remaining work as possible on machine two, and so forth. All the virtual machines will be simulated in parallel by the real processor.

Theorem 6. *The algorithm completes all tasks by their deadlines with speedup $\frac{\alpha}{\alpha-1}$.*

Proof. Let w_j be the total work performed by machine M_j, w_j^* be the total work OPT performs at speed s_j, and W be the total work in the instance. Note that our speedup is sufficient that we can run all machines 1 through j in parallel while paying only p_j per time. Thus our energy consumption is bounded by $\sum_j w_j p_j / s_j$ whereas OPT has energy consumption $\sum_j w_j^* p_j / s_j$. Since we push as much work onto the lowest virtual machine as possible, the amount of work we perform at the first k speeds is at least the amount of work OPT performs at the first k speeds:

$$\sum_{j=1}^{k} w_j^* \leq \sum_{j=1}^{k} w_j \Rightarrow \sum_{j=k+1}^{m} w_j = W - \sum_{j=1}^{k} w_j \leq W - \sum_{j=1}^{k} w_j^* = \sum_{j=k+1}^{m} w_j^*.$$

Using our assumption that $\frac{p_j}{s_j} \leq \frac{p_{j+1}}{s_{j+1}}$, it follows that we use at most as much energy as optimum. □

To compare our result against the work of Yao *et al.* [33], we consider the case where all speeds are available and $c(s) = s^p$. Their main results are a $2^p p^p$-competitive algorithm, along with an 8-competitive algorithm when $p = 2$. We improve these to 4^p and 6.5 respectively. The proof of the following theorem is in Appendix C.

Theorem 7. *We can obtain competitive ratio $(2^p - 1)^2/(p \ln 2)$ with continuous speeds and $c(s) = s^p$.*

5.1 Unrelated Tasks

We consider a modification of the task scheduling problem which permits tasks to react differently when the processor speed is increased. This models, for example, inputs where some tasks have memory as a primary bottleneck (thus they do not speed up much when the processor is accelerated) whereas other tasks are primarily computational (run-time inversely proportional to speed). The formal problem definition follows.

Problem 5 (Variable Workload Task Scheduling: VWTS). *A set of n tasks $(\vec{t_i}, a_i, d_i)$ arrives online, where $\vec{t_i}$ specifies task i's computation time at each of the speeds. We have one processor with m speeds s_j that use energy p_j. Find an order to execute tasks and a processor speed schedule where each task i runs only between its arrival time a_i and deadline d_i, and if i runs at s_j for time $\tau_i[j]$ then $\Sigma_j \frac{\tau_i[j]}{t_i[j]} = 1$. The goal is to minimize the total energy usage $\sum_i \sum_j p_j \tau_i[j]$.*

We observe that the offline version of VWTS can be solved optimally in polynomial time using a carefully designed linear program (see Appendix D). We establish a reduction between VWTS and the Packet Scheduling problem (PS) described in section 4. A packet scheduling algorithm is said to Decide-on-Arrival if the determination as to whether packet j will be dropped is made as soon as packet j arrives (and never changed at a later point in the algorithm). The algorithms described in Theorem 5 have this property.

Theorem 8. *A ρ-competitive algorithm with speedup s for VWTS implies a ρs-competitive algorithm with speedup s for PS. A ρ-competitive algorithm with speedup s for PS which is Decide-on-Arrival implies a ρ-competitive algorithm with speedup $O(s)$ for VWTS.*

Proof. Suppose we are given an algorithm for VWTS, and we want to solve PS. We pretend there are two speeds s_1 and s_2 which use energy 0 and P, respectively. Whenever packet (a_i, d_i, b_i) arrives, we suppose that a task has arrived with the same arrival time and deadline, which has $t_i[1] = 1$ and $t_i[2] = \frac{1}{P} b_i$. We assume P is extremely large, implying that we have no difficulty meeting deadlines at s_2. Our goal is now to *minimize* the total benefit of the tasks which we run at s_2, which is identical to minimizing the total benefit of tasks which we fail to run at s_1. This is identical to PS.

Now suppose we are given an algorithm for PS and want to solve VWTS. For each task i, define benefit $b_i[j] = t_i[j+1]p_{j+1} - t_i[j]p_j$. This represents the energy saved by running task i at s_j instead of s_{j+1}. Let X_j^* be the set of tasks which OPT runs at s_j; then OPT's energy usage is given by $\Sigma_j \Sigma_{i \in X_j^*} t_i[j] p_j = \Sigma_j \Sigma_{i \in X_k^*; k > j} b_i[j] + \Sigma_i t_i[1] p_1$. Thus if we can approximately minimize the sum of benefits, we can also approximately minimize the energy. We now model our system as j parallel machines, where machine j runs at s_j or 0. We divide each task into an appropriate number of packets with scaled benefits (*i.e.* $\frac{b_i[j]}{t_i[j]}$) and let each machine see the set of tasks not finished by slower machines. Let X_j be the set of tasks that are not finished on the first j machines by our algorithm. It follows that $\Sigma_{i \in X_j} b_i[j] \leq \rho \Sigma_{i \in X_k^*; k > j} b_i[j]$ since it is possible to finish all tasks that OPT does at speeds s_j or lower on one machine which runs always at s_j. Summing gives us a total energy usage of at most OPT. If we Decide-on-Arrival, our total speed is the sum of lower speeds times the speedup s, giving an $O(s)$ speedup when speeds scale geometrically. $\square$

Combining with our previous results, this implies an $O(\min\{\log V, \log W\})$-speedup optimum algorithm for VWTS, and an $O(1)$-speedup optimum algorithm for VWTS with agreeable deadlines. No constant-speedup, bounded-competitive algorithm is possible for VWTS in the general case.

References

[1] Susanne Albers. Energy-efficient algorithms. *Communications of the ACM*, 53:86–96, 2010.

[2] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *Lecture Notes in Computer Science*, 3884, 2006.

[3] Nikhil Bansal, Niv Buchbinder, Anupam Gupta, and Joseph Naor. An $O(\log^2 k)$-competitive algorithm for metric bipartite matching. In *Proceedings of the 13th European Symposium on Algorithms*, 2007.

[4] Nikhil Bansal, Ho-Leung Chan, Tak-Wah Lam, and Lap-Kei Lee. Scheduling for bounded speed processors. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 2008.

[5] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2009.

[6] Nikhil Bansal, Kirk Pruhs, and Clifford Stein. Speed scaling for weighted flow time. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[7] Yair Bartal, Francis Y. L. Chin, Marek Chrobak, Stanley P. Y. Fung, Wojciech Jawor, Ron Lavi, Jiří Sgall, and Tomáš Tichý. Online competitive algorithms for maximizing weighted throughput of unit jobs. In *Proceedings of the 21st Symposium on Theoretical Aspects of Computer Science*, 2004.

[8] Sanjoy Baruah, Jayant Haritsa, and Nitin Sharma. On-line scheduling to maximize task completions. In *IEEE Real-time Systems Symposium*, 1994.

[9] Luca Becchetti, Stefano Leonardi, Alberto Marchett-Spaccamela, and Kirk Pruhs. Online weighted flow time and deadline scheduling. In *Workshop on Approximation Algorithms for Combinatorial Optimization*, 2001.

[10] Enrico Bini, Giorgio Buttazzo, and Giuseppe Lipari. Minimizing CPU energy in real-time systems with discrete speed management. *ACM Transactions on Embedded Computer Systems*, 8(4):1–23, 2009.

[11] Marek Chrobak, Leah Epstein, John Noga, Jiří Sgall, Rob van Stee, Tomáš Tichý, and Nodari Vakhania. Preemptive scheduling in overloaded systems. *Lecture Notes in Computer Science*, 2380, 2002.

[12] Marek Chrobak, Wojciech Jawor, Jiří Sgall, and Tomáš Tichý. Improved online algorithms for buffer management in QoS switches. *Lecture Notes in Computer Science*, 3221, 2004.

[13] Béla Csaba and András Pluhár. A randomized algorithm for the on-line weighted bipartite matching problem. *Journal of Scheduling*, 11(6):449–455, 2008.

[14] Matthias Englert and Matthias Westermann. Considering suppressed packets improves buffer management in QoS switches. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[15] Bruce Hajek. On the competitiveness of on-line scheduling of unit-length packets with hard deadlines in slotted time. In *Proceedings of the Conference on Information Sciences and Systems*, pages 434–439, 2001.

[16] Bala Kalyanasundaram and Kirk Pruhs. On-line weighted matching. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, 1991.

[17] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4), 2000.

[18] Bala Kalyanasundaram and Kirk Pruhs. Maximizing job completions online. *Journal of Algorithms*, 49(1):63–85, 2003.

[19] Alexander Kesselman, Zvi Lotker, Yishay Mansour, Boaz Patt-Shamir, Baruch Schieber, and Maxim Sviridenko. Buffer overflow management in QoS switches. In *Proceedings of the 33rd annual ACM Symposium on Theory of Computing*, 2001.

[20] Samir Khuller, Stephen Mitchell, and Vijay Vazirani. On-line algorithms for weighted bipartite matching and stable marriages. *Theoretical Computer Science*, 127(2):255–267, 1994.

[21] Chiu-Yuen Koo, Tak-Wah Lam, Tsuen-Wan Ngan, and Kar-Keung To. On-line scheduling with tight deadlines. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science*, 2001.

[22] Gilad Koren, Dennis Shasha, and Shih-Chen Huang. MOCA: a multiprocessor on-line competitive algorithm for real-time system scheduling. *Theoretical Computer Science*, 128(1-2):75–97, 1994.

[23] Tak-Wah Lam, Lap-Kei Lee, Isaac K. To, and Prudence W. Wong. Speed scaling functions for flow time scheduling based on active job count. In *Proceedings of the 16th European Symposium on Algorithms*, 2008.

[24] Tak-Wah Lam and Kar-Keung To. Trade-offs between speed and processor in hard-deadline scheduling. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[25] Fei Li, Jay Sethuraman, and Clifford Stein. An optimal online algorithm for packet scheduling with agreeable deadlines. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2005.

[26] Fei Li, Jay Sethuraman, and Clifford Stein. Better online buffer management. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[27] Aranyak Mehta, Amin Saberi, Umesh Vazirani, and Vijay Vazirani. Adwords and generalized online matching. *Journal of the ACM*, 54(5), 2007.

[28] Adam Meyerson, Akash Nanavati, and Laura Poplawski. Randomized online algorithms for minimum metric bipartite matching. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2006.

[29] Intergovernmental Panel on Climate Change. Climate change 2007. *Fourth Assessment Report*, 2007.

[30] Kirk Pruhs, Jiří Sgall, and Eric Torng. Online scheduling. In *Handbook of Scheduling*. Chapman & Hall, 2004.

[31] Sebi Ryffel, Thanos Stathopoulos, Dustin McIntire, William Kaiser, and Lothar Thiele. Accurate energy attribution and accounting for multi-core systems. In *Technical Report 67, Center for Embedded Network Sensing*, 2009.

[32] Thanos Stathopoulos, Dustin McIntire, and William Kaiser. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *IPSN '08: Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, 2008.

[33] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, 1995.

A Lower Bounds for Benefit-Maximizing Task Scheduling

Recall that tasks arrive online and each is specified by a tuple (ℓ_i, a_i, d_i, v_i) where ℓ_i is the workload, a_i is the arrival time, d_i is the deadline, and v_i is the value. We will show that even in simple cases where only one non-zero speed is allowed and tasks are required to be identical in various characteristics, we still cannot obtain constant competitive algorithms. In fact, deterministic algorithms have polynomial lower bounds.

Theorem 9. *Even if $d_i - a_i = \ell_i = W$ for all tasks i, there is no constant competitive algorithm.*

Proof. Consider the set of tasks where $t_i = (W, \alpha^i, W + \alpha^i, W + \alpha^i)$, for $0 \le i < T$ and a speed structure where we can run only at speed $s = 1$ for cost 1 per unit time ($\hat{c}(1) = 1$) or at speed $s = 0$ for cost 0 per unit time ($\hat{c}(0) = 0$).

If we run a deterministic algorithm, only tasks t_0 and t_1 are relevant. We must start running on t_0 immediately, or task t_1 will never arrive and we will get an infinite competitive ratio. Once t_1 arrives, however, we can only get a total profit of 1, regardless of whether we switch tasks or not. OPT will ignore t_0 and start t_1 when it arrives, for a profit of α. As α must be smaller than W to enforce that only one task can be done, this gives us an $\Omega(W)$ polynomial deterministic lower bound.

We can do better with a randomized algorithm. If we bound $\alpha^i < W$, then we can still only do one task. If we do not choose the last-arriving task to do, we get at best a competitive ratio of α. The adversary continues with the tasks until one arrives which our algorithm has probability at most $\frac{1}{T}$ of executing. This gives a competitive ratio of at best $\Omega(\frac{\log W}{\log \log W})$. □

Theorem 10. *Even if $d_i - a_i = v_i = W$ for all tasks i, there is no constant competitive algorithm.*

Proof. Consider the set of tasks where $t_i = (W - \alpha^i, \alpha^i, W + \alpha^i, W)$, for $0 \le i < T$ and a speed structure where we can run only at speed $s = 1$ for cost 1 per unit time ($\hat{c}(1) = 1$) or at speed $s = 0$ for cost 0 per unit time ($\hat{c}(0) = 0$).

If we run a deterministic algorithm, only tasks t_0 and t_1 are relevant. We must start running on t_0 immediately, or task t_1 will never arrive and we will get an infinite competitive ratio. Once t_1 arrives, however, we can only get a total profit of 1, regardless of whether we switch tasks or not. OPT will ignore t_0 and start t_1 when it arrives, for a profit of α. As α must be smaller than $W/2$ to enforce that only one task can be done, this gives us an $\Omega(W)$ polynomial deterministic lower bound.

We can do better with a randomized algorithm. If we bound $2 < \alpha^i < W/2$, then we can still only do one task, and we must start each task before the next one arrives if we wish to complete it. If we do not choose the last-arriving task to do, we get at best a competitive ratio of α. The adversary continues scheduling tasks until one arrives which our algorithm has probability at most $\frac{1}{T}$ of executing. This gives a competitive ratio of at best $\Omega(\frac{\log W}{\log \log W})$. $\square$

Of course, if all tasks have identical workload, value, and window, then we can run the algorithm of [8] to obtain a constant-competitive result. The lower bounds imply that holding one or two of these properties fixed is insufficient, and hence our competitive ratio needs to be logarithmic in natural parameters.

B Algorithms for Packet Scheduling with Speedup

Lemma 6. *There is an online $O(\log W)$-speedup optimum algorithm for packet scheduling.*

Proof. We round arrival times and deadlines as follows. For each packet i, let $f_i = \lfloor \log_2(\frac{d_i - a_i + 1}{2}) \rfloor$ and g_i such that $a_i \leq 2^{f_i} g_i + 1$ and $2^{f_i}(g_i + 1) < d_i$. We let $a'_i = 2^{f_i} g_i + 1$ and $d'_i = 2^{f_i}(g_i + 1)$. We argue that a factor 3-speedup will allow us to send the same total value as optimum while satisfying these new arrival times and deadlines in lemma 7. We can now sort into $\log W$ buckets based on the value of f_i, and packets within each bucket have either identical or disjoint windows (the time interval from arrival to deadline). We will run on all buckets simultaneously and independently, making use of our overall speedup factor of $O(\log W)$. Within each bucket, we can easily solve optimally based on the modified arrival times. $\square$

Theorem 11 (Hall's Theorem)**.** *A bipartite graph $G = (V_1 \cup V_2, E)$ contains a ρ-to-one complete mapping from V_1 to V_2 iff $\forall X \subseteq V_1$, $\rho|\Gamma(X)| \geq |X|$ (where $\Gamma(X)$ denotes the neighborhood of X).*

Lemma 7. *If it was possible to complete some total value of packets by their original deadlines, then using a processor which always runs 3 times faster, it is possible to complete the same value between their modified arrival times a'_i and deadlines d'_i (as described in lemma 6).*

Proof. Suppose that it was possible to complete some subset of the packets by their original deadlines. We discretize time into increments, where at each increment optimum completes a single packet. We will establish a three-to-one mapping ϕ from packets to time increments, with the property that for every packet x sent by optimum we have $\phi(x) \in [a'_i, d'_i]$. Provided that such a mapping exists, we can complete all packets completed by optimum by making use of a factor 3 speedup. The key to the proof is thus showing that ϕ exists, for which we will apply Hall's theorem.

We need to prove that for any set of packets X completed by optimum, the set of feasible time steps $\Gamma'(X) = \bigcup_{x \in X} [a'_x, d'_x]$ between the modified arrivals and deadlines has size $|\Gamma'(X)| \geq \frac{1}{3}|X|$. Suppose for the sake of contradiction that X is the smallest set of packets completed by optimum where this does not hold. If $\Gamma'(X)$ contains several disjoint time intervals, then we can break X down into packets belonging to these intervals, one of which must also violate the inequality (implying that X is not the smallest set where it fails). Since this is a contradiction, we can assume that $\Gamma'(X) = [u+1, v]$ for some $u < v$. We now consider the original arrival times and deadlines for the packets $x \in X$. Due to the definition of the modified times, we can conclude that $a_x \geq 2u - v + 1$ and that $d_x \leq 2v - u$, from which we have a set of feasible times for X according to the original arrivals and deadlines of $\Gamma(X) \subseteq [2u - v + 1, 2v - u]$. We know that optimum can schedule all members of X between their true arrival and deadlines, so we have $|X| \leq |\Gamma(X)| \leq 3v - 3u$ by applying Hall's theorem again. But then $|\Gamma'(X)| = v - u \geq \frac{1}{3}|\Gamma(X)| \geq \frac{1}{3}|X|$, which shows that in fact our inequality is satisfied. We conclude that a processor which can send three packets per time step can send all packets sent by optimum between their modified arrival times and deadlines. $\square$

We say deadlines are *agreeable* if for every i, j with $[a_i, d_i] \subseteq [a_j, d_j]$, either $a_i = a_j$ or $d_i = d_j$ (*i.e.* later arrival implies later deadline). We observe that if we modify the arrival times and deadlines as described in lemma 7, the modified instance still has agreeable deadlines. Packets with $f_i = 0, 1, 2$ can be scheduled optimally with a 9-speedup. For packets with $f_i \geq 3$, we further modify the arrival times and deadlines setting $a''_i = (2^{f_i})g_i + (3/4)2^{f_i - 1} + 1$ and $d''_i = (2^{f_i})(g_i + 1) - (3/4)2^{f_i-1}$. Using Hall's theorem again, we see that a 12-speedup can still achieve optimum benefit. In total, a 21-speedup is sufficient, provided we can prove the following lemma:

Lemma 8. *There is an online algorithm which solves packet scheduling optimally on the modified arrival and deadline times a''_i, d''_i provided the original arrivals and deadlines were agreeable.*

Proof. Consider two packets i, j where $[a''_i, d''_i]$ intersects $[a''_j, d''_j]$. Then since $[a''_i, d''_i] \subseteq [a'_i, d'_i]$ (and analogously for j), we can conclude that either $a'_i = a'_j$ or $d'_i = d'_j$. We will consider the case when the arrival times are the same (the other case is symmetric).

Since $a'_i = a'_j$ then $2^{f_i} g_i = 2^{f_j} g_j$. Thus, if $f_i = f_j$ then $[a''_i, d''_i] = [a''_j, d''_j]$. Otherwise, assume without loss of generality that $f_i < f_j$. Then:

$$a''_j = 2^{f_j} g_j + 1 + (3/4)2^{f_j - 1} \geq 2^{f_i} g_i + 1 + (6/4)2^{f_i-1} > 2^{f_i}(g_i + 1) - (3/4)2^{f_i-1} = d''_i$$

But this contradicts the fact that $[a''_i, d''_i]$ and $[a''_j, d''_j]$ intersect. Thus, all packets have either disjoint or identical intervals. Such an instance can be solved optimally in a greedy fashion. $\square$

Lemma 9. *There is an online $O(\log V)$-speedup optimum algorithm for packet scheduling.*

Proof. We partition the packets into $\log V$ buckets, based on each packet i's value b_i. We will use the speedup to send as many as two packets from each bucket at each time step. For each bucket j, we will greedily construct a set of packets P_j. Each time a packet p arrives, we determine whether it is possible to send (offline) all the packets of $P_j \cup \{p\}$, at most two per time step, by matching them to times between arrival and deadline. If so, we add p to P_j. It should be clear that the final P_j is maximal. In terms of actually transmitting packets, at each step we transmit the two packets in the current set P_j which have the earliest deadline. It is not hard to show by a swapping argument that this will succeed in transmitting all packets of P_j.

Let P^*_j be the set of packets from bucket j which are transmitted by optimum. We construct a graph where the vertices are the packets of $P_j \cup P^*_j$, with a directed edge (p, q) if packet $p \in P^*_j$ and if packets p and q were transmitted at the same time by our algorithm and by optimum. We observe that each node has in-degree at most one, since at most one optimum packet from P^*_j was transmitted at any time step, and that nodes representing packets from $P^*_j - P_j$ have in-degree zero. For each packet $p \in P^*_j - P_j$, we consider the subgraph consisting of nodes reachable from p. Note that these subgraphs are disjoint and acyclic because all nodes have in-degree at most one. Packet p must have two outgoing edges, since otherwise our algorithm sent at most one packet at the time when optimum sent p, and we could therefore augment P_j by adding the single packet p to it. Thus we conclude that the component of p includes at least two nodes q_1, q_2 with out-degree zero. If one of these nodes were a member of P^*_j, then we consider the path from $p = x_0, x_1, x_2, ..., x_k = q_i$. Each of these packets is sent at some time by optimum, and all but the first are sent by our algorithm as well. We could modify our algorithm's output to send each x_i at the time when the optimum sends x_i; this keeps the number of packets sent at each time the same, except for the time when optimum sends $x_k = q_i$... but since this node has out-degree zero, it follows that our algorithm was not sending *anything* at that time, and thus remains feasible. We conclude that we could add p to P_j, contradicting local maximality. Thus nodes q_1, q_2 must not be members of P^*_j. So for each packet in $P_j^* - P_j$, there are at least two corresponding packets in $P_j - P_j^*$. We then measure the total benefit:

$$b(P_j) \geq b(P_j \cap P_j^*) + 2^j|P_j - P_j^*| \geq b(P_j \cap P_j^*) + 2^{j+1}|P_j^* - P_j| \geq b(P_j^*)$$

Since the total benefit of P_j is at least the total benefit of P_j^*, we conclude that our algorithm's total benefit *sent* is at least optimum. Thus the total benefit *lost* by our algorithm is at most that of optimum. So with speedup $2\log V$ we have obtained optimum Packet Scheduling. $\square$

C Proof of Theorem 7

Since the problem calls for continuous speeds, we can select any discrete speeds we like. We will use $s_j = \alpha^{r+j}$ for a chosen $r \in [0,1]$. The description of the algorithm will assume that r is selected uniformly at random; we can de-randomize this by simultaneously running multiple copies of the algorithm for each r which is a multiple of some small ϵ, then allowing each copy to handle an ϵ fraction of the workload of each task.

Note that the optimum solution can run at any speed, whereas our solution is assumed to run only at speeds s_j; this will cause us to lose some factor in the competitive ratio. Consider some speed $s_{OPT} = q\alpha^j$ used by optimum; we will replace this by the lowest speed $s_j \geq s_{OPT}$ for an expected extra energy required of:

$$E[energy] = \int_0^{\log_\alpha q} \frac{1}{q^p}\alpha^{rp+p}dr + \int_{\log_\alpha q}^1 \frac{1}{q^p}\alpha^{rp} = \frac{\alpha^p - 1}{p\ln\alpha}$$

Thus in expectation, the restricted-speed version of the problem can be solved using at most this factor increase in energy. Of course, we cannot compute the optimum solution even for the restricted-speed version online. However, our algorithm guarantees that we would use at most the energy for the restricted-speed optimum provided that we could simultaneously run the virtual machines M_1 through M_j (for any j) by using energy $\Sigma_{i=1}^j c^p\alpha^{pi}$. Of course, the real cost of simultaneously running these virtual machines is $(\Sigma_{i=1}^j c\alpha^i)^p$. The ratio between these will be largest when j is large, so considering the limit as j goes to infinity we are comparing an energy of $c^p\frac{\alpha^p}{\alpha^p-1}$ against $c^p(\frac{\alpha}{\alpha-1})^p$. Dividing and canceling appropriately gives a ratio of $\frac{\alpha^p-1}{(\alpha-1)^p}$. Multiplying the extra energy needed to discretize the speeds by the extra energy needed to simulate the virtual machines gives competitive ratio $\frac{(\alpha^p-1)^2}{p(\alpha-1)^p\ln\alpha}$. Setting $\alpha = 2$ gives the result claimed in theorem 7.

D Offline task scheduling with variable workloads

In this section, we show how to optimally solve the offline version of the task scheduling with variable workloads problem. In this case, tasks take the form of a triplet $(\vec{t_i}, a_i, d_i)$ where $\vec{t_i}$ specifies the computation times at each of the speeds. Our job is to determine the tasks being performed and the speeds at which to run at each point in time so as to minimize energy while completing all tasks by their deadlines. We show how to do this using linear programming.

Consider a task $(\vec{t_i}, a_i, d_i)$. Recall that t_{ij} denotes the time it takes to complete task i at speed j and that $t_{i1} \geq \cdots \geq t_{im}$. However, our ability to split a task across multiple speeds gives us a continuum of computation times for task i. In particular, for every pair of speeds $s_j, s_{j'}$ and $\alpha \in [0,1]$, we can complete task i by running it for αt_{ij} seconds at speed s_j and $(1-\alpha)t_{ij'}$ seconds at speed $s_{j'}$. The energy required to do this is simply $\alpha p_j t_{ij} + (1-\alpha)p_{j'}t_{ij'}$.

Thus, we can achieve any computation time within $[t_{i1}, t_{im}]$. Suppose for some speed s_j, there were two speeds s_a and s_b such that $s_a < s_j < s_b$ and such that when $\alpha = \frac{t_{ij}-t_{ib}}{t_{ia}-t_{ib}}$ we have

$$p_j t_{ij} \geq \alpha p_a t_{ia} + (1-\alpha) p_b t_{ib}.$$

Then it is in fact more energy efficient to perform this task using a linear combination of speeds s_a and s_b than it is to perform the task at speed s_j. Thus, in this case s_j is non-optimal and we should never use this speed for task i. We can easily identify the optimal speeds $s_1^i, \ldots, s_{m_i}^i$ using a convex hull calculation. Now, given a desired computation time y for a task i, the minimum amount of energy we must use in order to finish i in exactly y time is given by the convex function:

$$e_i(y) = \begin{cases} \frac{p_1 t_{i1} - p_2 t_{i2}}{t_{i1} - t_{i2}}(y - t_{i2}) + p_2 t_{i2}, & \text{if } t_{i1} \leq y < t_{i2}, \\ \frac{p_2 t_{i2} - p_3 t_{i3}}{t_{i2} - t_{i3}}(y - t_{i3}) + p_3 t_{i3}, & \text{if } t_{i2} \leq y < t_{i3}, \\ \vdots & \\ \frac{p_{m_i-1} t_{i,m_i-1} - p_{m_i} t_{i,m_i}}{t_{i,m_i-1} - t_{i,m_i}}(y - t_{i,m_i}) + p_{m_i} t_{i,m_i}, & \text{if } t_{i,m_i-1} \leq y < t_{i,m_i}. \end{cases}$$

We now use linear programming in the follwing way: Let $c_0 < c_1 < \cdots < c_T$ be the times at which some task arrives or is due. We use variables x_{ik} to denote the amount of time between c_k and c_{k+1} spent on task i, y_i to indicate the total computation time of task i, and e_i for the total energy used to compute task i in t_i time. Our LP is as follows:

$$\begin{array}{lll} \text{minimize:} & \sum_i e_i + p_1 \sum_k (c_{k+1} - c_k - \sum_i x_{ik}) & \\ \text{subject to:} & e_i \geq \frac{p_j t_{ij} - p_{j+1} t_{i,j+1}}{t_{ij} - t_{i,j+1}}(y_i - t_{i,j+1}) + p_{j+1} t_{i,j+1} & \forall i, \forall j \in \{1, \ldots, m_i\} \\ & x_{ik} = 0 & \forall i, \forall k \text{ with } c_k < a_i \text{ or } c_k \geq d_i \\ & y_i = \sum_k x_{ik} & \forall i \\ & t_{i1} \leq y_i \leq t_{i,m_i} & \forall i \\ & \sum_i x_{ik} \leq c_{k+1} - c_k & \forall k \end{array}$$

Notice here that our objective function minimizes the total energy used among all tasks (and also assumes that during idle times we run at the lowest energy speed s_1). Our first constraint uses the convex function described above to calculate the energy of task i. Our second constraint ensures that we never perform a task before its arrival or after its deadline. Our third constraint calculates the total computation time of our task. The fourth constraint ensures that our total computation time is feasible. Our final constraint ensures that we do not schedule too much work between each c_k.

After solving this LP, we can use the x_{ik} and y_i variables to reconstruct our schedule. We use y_i to determine the speeds at which to run our task and the fraction of the task we should run at each speed. We use the x_{ik} variables to determine when we should be performing which tasks (within c_k and c_{k+1} the order in which tasks are performed does not affect solution quality). It is straightforward to combine these two to completely recover our energy-efficient schedule.