J Sched (2009) 12: 489–500
DOI 10.1007/s10951-009-0123-y

Power-aware scheduling for makespan and flow

David P. Bunde

Published online: 29 July 2009

Abstract We consider offline scheduling algorithms that incorporate speed scaling to address the bicriteria problem of minimizing energy consumption and a scheduling metric. For makespan, we give a linear-time algorithm to compute all non-dominated solutions for the general uniprocessor problem and a fast arbitrarily-good approximation for multiprocessor problems when every job requires the same amount of work. We also show that the multiprocessor problem becomes NP-hard when jobs can require different amounts of work.

For total flow, we show that the optimal flow corresponding to a particular energy budget cannot be exactly computed on a machine supporting exact real arithmetic, including the extraction of roots. This hardness result holds even when scheduling equal-work jobs on a uniprocessor. We do, however, extend previous work by Pruhs et al. to give an arbitrarily-good approximation for scheduling equal-work jobs on a multiprocessor.

Keywords Power-aware scheduling · Dynamic voltage scaling · Speed scaling · Makespan · Total flow

1 Introduction

Power consumption is becoming a major issue in computer systems. This is most obvious for battery-powered systems such as laptops because processor power consumption has been growing much more quickly than battery capacity. Even systems that do not rely on batteries have to deal with power consumption since nearly all the energy consumed by a processor is released as heat. The heat generated by modern processors is becoming harder to dissipate and is particularly problematic when large numbers of them are in close proximity, such as in a supercomputer or a server farm. The importance of the power problem has led to a great deal of research on reducing processor power consumption; see overviews by Mudge (2001), Brooks et al. (2000), and Tiwari et al. (1998). We focus on the technique *dynamic voltage scaling*, which allows the processor to enter low-voltage states. Reducing the voltage reduces power consumption, but also forces a reduction in clock frequency so the processor runs more slowly. For this reason, dynamic voltage scaling is also called *frequency scaling* and *speed scaling*.

This paper considers how to schedule processors with dynamic voltage scaling so that the scheduling algorithm determines how fast to run the processor in addition to choosing a job to run. In classical scheduling problems, the input is a series of n jobs $J_1, J_2, \ldots, J_n$. Each job J_i has a *release time* r_i, the earliest time it can run, and a *processing time* p_i, the amount of time it takes to complete. With dynamic voltage scaling, the processing time depends on the schedule so instead each job J_i comes with a *work requirement* w_i. We use p_i^A to denote the processing time of job J_i in schedule A. A processor running continuously at speed σ completes σ units of work per unit of time so job J_i would have processing time w_i/σ. In general, a processor's speed is a function of time and the amount of work it completes is the integral of this function over time. This paper considers *offline* scheduling, meaning the algorithm receives all the input together. This is in contrast to *online* scheduling, where the algorithm learns about each job at its release time.

A preliminary version of this work was presented at the 18th ACM Symposium on Parallelism in Algorithms and Architectures (Bunde 2006).

Partially supported by NSF grant CCR 0093348.

D.P. Bunde (✉)
Department of Computer Science, Knox College, Galesburh, USA
e-mail: dbunde@knox.edu

To measure schedule quality, we use two classic metrics. Let S_i^A and C_i^A denote the start and completion times of job J_i in schedule A. Most of the paper focuses on minimizing the schedule's *makespan*, $\max_i C_i^A$, the completion time of the last job. We also consider *total flow*, the sum over all jobs of $C_i^A - r_i$, the time between the release and completion times of job J_i.

Either of these metrics can be improved by using more energy to speed up the last job so the goals of low energy consumption and high schedule quality are in opposition. Thus, power-aware scheduling is a bicriteria optimization problem and our goal becomes finding *non-dominated schedules* (also called *Pareto optimal schedules*), such that no schedule can both be better and use less energy. A common approach to bicriteria problems is to fix one of the parameters. In power-aware scheduling, this gives two interesting special cases. If we fix energy, we get the *laptop problem*, which asks "What is the best schedule achievable using a particular energy budget?" Fixing schedule quality gives the *server problem*, which asks "What is the least energy required to achieve a desired level of performance?"

To calculate the energy consumed by a schedule, we need a function relating speed to power; the energy consumption is then the integral of power over time. Actual implementations of dynamic voltage scaling give a list of speeds at which the processor can run. For example, the AMD Athlon 64 can run at 2000, 1800, or 800 MHz (Advanced Micro Devices 2004). Since the first work on power-aware scheduling algorithms (Weiser et al. 1994), however, researchers have assumed that the processor can run at an arbitrary speed within some range. The justification for allowing a continuous range of speeds is twofold. First, choosing the speed from a continuous range is an approximation for a processor with a large number of possible speeds. Second, a continuous range of possible clock speeds is observed by individuals who use special motherboards to overclock their computers.

Most power-aware scheduling algorithms use the model proposed by Yao et al. (1995), in which the processor can run at any non-negative speed and power $=$ speed$^\alpha$ for some constant $\alpha > 1$. In this model, the energy required to run job J_i at speed σ is $w_i \sigma^{\alpha-1}$ since the running time is w_i/σ. This relationship between power and speed comes from an approximation of a system's switching loss, the energy consumed by logic gates switching values. The so-called cube-root rule for this term suggests the value $\alpha = 3$ (Brooks et al. 2000).

Most of our results hold in generalizations of the model power $=$ speed$^\alpha$. Specifically, we consider continuous power functions that are convex or strictly convex, as well as discrete power functions. A function is *convex* if the line segment between any two points on its curve lies on or above the curve. A function is *strictly convex* if the line segment lies strictly above the curve except at its endpoints. The power function power $=$ speed$^\alpha$ is convex when $\alpha = 1$ and strictly convex when $\alpha > 1$. Throughout, we assume that running at speed 0 requires no energy. The power consumption of real processors includes a component that is independent of the processing speed, but this overhead can be deducted from the energy budget given to our algorithm.

This paper considers both uniprocessor and multiprocessor scheduling. In the multiprocessor setting, we assume that the processors have a shared energy supply. This corresponds to scheduling a laptop with a multi-core processor or a server farm concerned only about total energy consumption and not the consumption of each machine separately.

Results Our results in power-aware scheduling are the following:

- For uniprocessor makespan, we give an algorithm to find all non-dominated schedules. Its running time is linear once the jobs are sorted by arrival time. This algorithm works even for discrete power functions.
- We show that there is no exact algorithm for uniprocessor total flow using exact real arithmetic, including the extraction of kth roots. This holds even with equal-work jobs.
- For a large class of reasonable scheduling metrics, we show how to extend uniprocessor algorithms to the multiprocessor setting with equal-work jobs. Using this technique, we give arbitrarily-good approximations for multiprocessor makespan of equal-work jobs and multiprocessor total flow of equal-work jobs.
- We prove that multiprocessor makespan is NP-hard if jobs require different amounts of work, even if all jobs arrive at the same time.

For the problems we consider, reordering jobs does not change solution quality. Thus, our results hold whether or not the scheduler can use *preemption*, pausing a job mid-execution and resuming it later. Our multiprocessor results assume that jobs cannot migrate between processors, however.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 gives the uniprocessor algorithm for makespan. Section 4 shows that total flow cannot be exactly minimized. Section 5 extends the uniprocessor results to give multiprocessor algorithms for equal-work jobs and shows that general multiprocessor makespan is NP-hard. Finally, Sect. 6 discusses future work.

2 Related work

The area of power-aware scheduling has recently attracted a lot of interest. Therefore, we restrict our discussion of related work to include only the most closely-related results,

focusing on those that minimize makespan or total flow. For a discussion of other metrics and other power-reduction techniques in scheduling, we refer the reader to the survey of Irani and Pruhs (2005). Similar problems have also been studied for industrial and commercial applications under the name *controllable processing times*. In these applications, instead of allocating power to speed up a processor, jobs are sped up by allocating resources like manpower, money, or fuel. See Shabtay and Steiner (2007) for a survey of these results. Most of them use models with a different power function or without release times, but some relevant work is cited below.

Minimizing makespan The work most closely related to ours is due to Uysal-Biyikoglu et al. (2002), who consider the problem of minimizing the energy of wireless transmissions. The only assumption required by their algorithms is that the power function is continuous and strictly convex. They give a quadratic-time algorithm to solve the server version for makespan. Thus, our algorithm represents an improvement by running faster, working for more general power functions, and finding all non-dominated schedules rather than just solving the server problem.

Several variations of the wireless transmission problem have also been studied. El Gamal et al. (2002) consider the possibility of packets with different power functions. They give an iterative algorithm that converges to an optimal solution. They also show how to extend their algorithm to handle the case when the buffer used to store active packets has bounded size and the case when packets have individual deadlines. Keslassy et al. (2003) claim a non-iterative algorithm for packets with different power functions and individual deadlines when the inverse of the power function's derivative can be represented in closed form. (Their paper gives the algorithm, but only a sketch of the proof of correctness.)

Another transmission scheduling problem, though one that does not correspond to a processor scheduling problem, is to schedule multiple transmitters. If only one transmitter can operate at a time, another extension of the iterative algorithm of Uysal-Biyikoglu et al. (2002) converges to the optimal solution. In general, however, there may be a better solution in which transmitters sometimes deliberately interfere with each other. Uysal-Biyikoglu and El Gamal (2004) give an iterative algorithm to find this solution.

Shabtay and Kaspi (2006) prove that the multiprocessor problem is NP-hard when the time to complete job J_i is $(w_i/x_i)^k$ where k is a constant and x_i is the amount of resource allocated to job J_i, with $\sum x_i$ bounded. Our proof of the NP-hardness of general multiprocessor scheduling is essentially the same as theirs, with the minor observation that the argument works for all strictly convex power functions. When the processing time of job J_i is $(w_i/x_i)^k$ and all jobs arrive at the same time, Shabtay and Kaspi (2006) also give algorithms for multiprocessor scheduling if the jobs are already assigned to processors or preemption is allowed.

Shakhlevich and Strusevich (2006) study a variation in which all jobs run at the same speed, which corresponds to buying a faster processor. They give an algorithm with running time $O(n \log n)$ to minimize the sum of makespan and processor cost in this setting. They also consider problems where the release times can be made earlier (at a cost) and individual processing times can be reduced at a cost linear in the amount of time saved. For this power function, the same authors give an $O(n \log n)$ time algorithm to solve the uniprocessor bicriteria problem (Shakhlevich and Strusevich 2005). They give a similar algorithm for the parallel bicriteria problem without release times.

In the processor scheduling literature, the work most closely related to the algorithms in this paper is due Pruhs et al. (2005). They consider the laptop problem version of minimizing makespan for jobs having precedence constraints where all jobs are released immediately and power = speed$^\alpha$. Their main observation, which they call the *power equality*, is that the sum of the powers of the machines is constant over time in the optimal schedule. They use binary search to determine this value and then reduce the problem to scheduling on related fixed-speed machines. Previously-known (Chudak and Shmoys 1997; Chekuri and Bender, 2001) approximations for the related fixed-speed machine problem then give an $O(\log^{1+2/\alpha} m)$-approximation for power-aware makespan. This technique cannot be applied in our setting because the power equality does not hold for jobs with release dates.

Minimizing the makespan of tasks with precedence constraints has also been studied in the context of project management. Speed scaling is possible when additional resources can be used to shorten some of the tasks. Pinedo, (2005) gives heuristics for some variations of this problem.

Minimizing flow time Power-aware schedule to minimize total flow time was first studied by Pruhs et al. (2004), who consider scheduling equal-work jobs on a uniprocessor. In this setting, they observe that jobs can be run in order of release time and then prove the following relationships between the speed of each job in the optimal solution:

Theorem 1 (Pruhs et al. 2004) *Let $J_1, J_2, \ldots, J_n$ be equal-work jobs ordered by release time. In the schedule* OPT *minimizing total flow time for a given energy budget where power = speed$^\alpha$, the speed σ_i of job J_i (for $i \neq n$) obeys the following*:

- *If $C_i^{\text{OPT}} < r_{i+1}$, then $\sigma_i = \sigma_n$.*
- *If $C_i^{\text{OPT}} > r_{i+1}$, then $\sigma_i^\alpha = \sigma_{i+1}^\alpha + \sigma_n^\alpha$.*
- *If $C_i^{\text{OPT}} = r_{i+1}$, then $\sigma_n^\alpha \leq \sigma_i^\alpha \leq \sigma_{i+1}^\alpha + \sigma_n^\alpha$.*

These relationships, together with observations about when the schedule changes configuration, give an algorithm based on binary search that finds an arbitrarily-good approximation for either the laptop or the server problem.

The algorithm of Pruhs et al. (2004) actually gives more than the schedule for a single energy budget. It can be used to plot the exact trade-off between total flow time and energy consumption for optimal schedules in which the third relationship of Theorem 1 does not hold. Their paper (Pruhs et al. 2004) includes such a plot with gaps where this relationship holds, i.e. where the optimal solution has one job completing exactly as another is released. Our impossibility result in Sect. 4 shows that the difficulty caused by the third relationship cannot be avoided.

Albers and Fujiwara (2006) propose a variation with the objective of minimizing the sum of energy consumption and total flow. When power = speed$^\alpha$, they show that every online nonpreemptive algorithm is $\Omega(n^{1-1/\alpha})$-competitive using an input instance where a short job arrives once the algorithm starts a long job. Their main result is an online algorithm for the special case of equal-work jobs whose competitive ratio is at most $8.3e(1+\phi)^\alpha$, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the Golden Ratio. This competitive ratio is constant for fixed α, but very large; for $\alpha = 3$, its value is approximately 405. They also give an arbitrarily-good approximation for the offline problem with equal-work jobs and suggest another possible online algorithm. Bansal et al. (2007) analyze this suggested online algorithm using a potential function and show it is 4-competitive. They also show that a related algorithm has competitive ratio around 20 for weighted jobs.

Other related works We conclude the discussion of related work by mentioning a couple of papers on minimizing the energy consumption of jobs with deadlines that have similarities to our work.

Although most work on power-aware scheduling assumes a continuous power function, we are not the first to consider discrete power functions. Chen et al. (2005) show that minimizing energy consumption in this setting while meeting all deadlines is NP-hard, but give approximations for some special cases.

In addition, Albers et al. (Albers et al. 2007) give a technique similar to our extension of uniprocessor algorithms to the multiprocessor setting that works for jobs with deadlines. Their specific result is briefly described in Sect. 5.

3 Makespan scheduling on a uniprocessor

Our first result is an algorithm to find all non-dominated schedules for uniprocessor power-aware makespan. We begin by solving the laptop problem for an energy budget E with a strictly-convex power function.

3.1 Algorithm for laptop problem

To find an optimal solution, we establish properties it must satisfy. Our first property allows us to fix the order in which jobs are run. To simplify notation, we assume the jobs are indexed so that $r_1 \le r_2 \le r_3 \le \ldots \le r_n$.

Lemma 2 *There is an optimal solution that runs jobs in order of their release times.*

Proof We show that any schedule can be modified to run jobs in order of their release times without changing the energy consumption or makespan. If schedule A is not in this form, then A runs some job J_i followed immediately by some job J_j with $j < i$. We change the schedule by starting job J_j at time S_i^A and starting job J_i after job J_j completes at time $S_i^A + p_j^A$. The speed of each job is the same as in schedule A so the energy consumption is unchanged. The interval of time when jobs J_i and J_j are running is also unchanged so the transformation does not affect makespan. The resulting schedule A' is legal since each job starts no earlier than its release time. In particular, $r_j \le r_i \le S_i^A = S_j^{A'}$ and $r_i \le S_i^A < S_i^{A'}$. □

The second property of optimal schedules is due to Yao et al. (1995), who observed that the optimal schedule does not change speed during a job or energy could be saved by running that job at its average speed.

Lemma 3 (Yao et al. 1995) *If the power function is strictly convex and* OPT *is an optimal schedule, then* OPT *runs each job at a single speed.*

This follows from the convexity of the power function and holds even if the number of speed changes can be infinite; it can be shown using Jensen's Inequality (cf. Rudin, 1987, p. 62). We use σ_i^A to denote the speed of job J_i in schedule A, omitting the schedule when it is clear from context.

The third property is that optimal schedules do not include idle time.

Lemma 4 *If the power function is strictly convex and* OPT *is an optimal schedule, then* OPT *is not idle between the release of job* J_1 *and the completion of all jobs.*

Proof Suppose to the contrary that OPT includes some idle time. If OPT is idle before running its first job, modify the schedule to run job J_1 during this idle time in addition to whenever job J_1 runs during OPT. Otherwise, there is a job J_i running before the idle time. In this case, slow down a job J_i so that it completes at the end of the idle time. In either case, our modification means some job runs more slowly.

This change saves energy, which can be used to speed up the last job and lower the makespan, contradicting the optimality of OPT. □

Stating the next property requires a definition. A *block* is a maximal substring of jobs such that each job except the last finishes after the arrival of its successor. For brevity, we denote a block with the indices of its first and last jobs. Thus, the block with jobs $J_i, J_{i+1}, \ldots, J_{j-1}, J_j$ is block (i, j). The fourth property is the analog of Lemma 3 for blocks.

Lemma 5 *If the power function is strictly convex and B is the set of jobs belonging to a block of an optimal schedule* OPT *then* OPT *runs every job in B at the same speed.*

To prove this lemma, we use a procedure that we call *speed swapping*. To use this procedure, we specify two jobs, J_i and J_j, plus a value $\epsilon > 0$ corresponding to an amount of work. The procedure modifies the schedule by swapping the speed at which ϵ work of each job is run. Specifically, it runs ϵ work from job J_i at speed σ_j and ϵ work from job J_j at speed σ_i. Any jobs running between jobs J_i and J_j have their start and completion times adjusted so that each job starts at the completion of its predecessor. In order to use speed swapping, we must argue that this sliding does not cause any job to start before its release time. Once we prove this, however, speed swapping gives us a way to change the schedule without affecting either the makespan or the total energy consumption; neither is changed since the modified schedule has the same amount of work running at each speed. In particular, using speed swapping on an optimal schedule gives another optimal schedule.

Using this property of speed swapping, we prove Lemma 5 by contradiction.

Proof of Lemma 5 If the lemma does not hold, we can find two adjacent jobs, J_i and J_j, in the same block of OPT with $\sigma_i \neq \sigma_j$. Let ϵ be a positive number less than the amount of work remaining in job J_i at time r_j. Construct a new schedule by speed swapping ϵ work between jobs J_i and J_j. By our choice of ϵ, the new schedule does not violate any release times. As discussed above, it is another optimal schedule. This contradicts Lemma 3 since job J_i does not run at a constant speed. □

Lemma 5 shows that speed is a property of blocks. In fact, if we know how an optimal schedule satisfying Lemma 2 divides jobs into blocks, we can compute the speed of each block. The definition of a block and Lemma 4 mean that block (i, j) starts at time r_i. Similarly, block (i, j) completes at time r_{j+1} unless it is the last block. Thus, any block (i, j) other than the last runs at speed $(\sum_{k=i}^{j} w_k)/(r_{j+1} - r_i)$. To compute the speed of the last block, we subtract the energy used by all the other blocks from the energy budget E. We choose the speed of the last block to exactly use the remaining energy.

Using the first four properties, we can use dynamic programming to compute the optimal schedule. Specifically, we fill in a table T, where $T[i]$ $(1 \leq i < n)$ is the minimum energy needed to complete jobs $J_1, \ldots, J_i$ by time r_{i+1}. Each $T[i]$ is computed as the minimum over $j < i$ of $T[j]$ and the cost of block $(j + 1, i)$. Once this table is filled, the minimum makespan is the earliest time block $(j + 1, n)$ that can be completed using energy $E - T[j]$ over all possible values of j. The only subtlety is that not all blocks are possible; unit-work jobs released at times 0 and 90 cannot be a single block completing at time 100 since the implied block speed of $100/2 = 50$ causes the first job to complete before the second is released. To avoid considering illegal blocks, we calculate a maximum speed $m_{(i,j)}$ for each possible block (i, j) using the relationship $m_{(i,j)} = \min\{m_{(i,j-1)}, \sum_{k=i}^{j} w_k/(r_{j+1} - r_i)\}$. Blocks whose speed exceeds their maximum are treated as having infinite cost when computing table entries.

A careful implementation of this algorithm runs in $O(n^2)$ time. To obtain a faster algorithm, we establish the following additional property:

Lemma 6 *If the power function is strictly convex and* OPT *is an optimal schedule, then the block speeds in* OPT *are non-decreasing.*

Proof Suppose to the contrary that OPT runs a block faster than the block following it. Let J_i be the job run at the end of the faster block and job J_j be the job beginning the slower block. We create a new schedule by speed swapping between jobs J_i and J_j, choosing ϵ to be less than the work of either job. The modified schedule is valid since the only start time modified is that of job J_j, which starts later than in OPT. (This is where we use that the earlier block is faster since otherwise the new schedule speeds up job J_i and finishes it before r_j.) Thus, we have created an optimal schedule that runs jobs J_i and J_j at two speeds, contradicting Lemma 3. □

It turns out that, for any level of energy consumption, only one schedule has all of the properties attributed to an optimal schedule in Lemmas 2–6. We state this result with the additional property that the last job runs as fast as possible. This property means that the energy consumption is the energy budget and also makes the lemma useful for power functions that are not strictly convex.

Lemma 7 *If the power function is strictly convex, there is a unique schedule having the following properties for any energy budget*:

1. *Jobs are run in order of release time.*

2. *Each job runs at a single speed.*
3. *The processor is not idle between the release of job J_1 and the completion of job J_n.*
4. *Jobs in each block run at the same speed.*
5. *The block speeds are non-decreasing.*
6. *The last block runs at the fastest speed allowed by the remaining energy.*

If the power function is convex, distinct schedules with these properties have the same blocks except that the last block of the higher-makespan schedule is the union of more than one block of the lower-makespan schedule.

Proof Suppose that A and B are different schedules with the listed properties and consuming the same amount of energy. With property 6, each schedule is determined by its blocks, so A and B must have different blocks. Without loss of generality, suppose the first difference occurs when job J_i is the last job in its block for schedule A but not for schedule B. We claim that every job indexed at least i runs slower and finishes later in schedule B than in schedule A.

First, we show this holds for job J_i. Job J_i ends its block in schedule A but not in schedule B, so $C_i^B > r_{i+1} = C_i^A$. Since each schedule begins the block containing job J_i at the same time and runs the same jobs before job J_i, job J_i runs slower in schedule B than schedule A.

Now we assume that the claim holds for jobs indexed below j and consider job J_j. Since each job $J_i, \ldots, J_{j-1}$ finishes no earlier than its successor's release time in schedule A, each finishes after its successor's release time in schedule B. Thus, none of these jobs ends a block in schedule B and schedule B places jobs J_i and J_j in the same block, which implies $\sigma_j^B = \sigma_i^B$. Speed is non-decreasing in schedule A, so $\sigma_i^A \le \sigma_j^A$. Therefore $\sigma_j^B = \sigma_i^B < \sigma_i^A \le \sigma_j^A$, so job J_j runs slower in schedule B than in schedule A. Job J_j also finishes later because job J_{j-1} finishing later implies that job J_j starts later.

If the power function is strictly convex, then energy consumption increases with speed and our claim implies that schedule B uses less energy than schedule A, a contradiction; so there must not be two such schedules. Even if the power function is not strictly convex, the claim means that B has higher makespan than A. In addition, we argued above that schedule B places job J_i in the last block. Thus, the last block of schedule B contains at least two blocks of schedule A, the one ending with job J_i and the one starting with job J_{i+1}. □

Because only an optimal schedule has the listed properties if the power function is strictly convex, we can solve the laptop problem by finding a schedule with all of them. For this task, we give the following algorithm IncMerge, which incrementally adds the jobs to a list L of blocks:

```
IncMerge(list of jobs J_1, J_2, ..., J_n sorted by release time)
1    L ← ∅
2    for i ← 1 to n
3        create block B consisting of job J_i
4        while L is nonempty and
             speed(B) < speed(last(L))
5            remove last(L)
6            add its jobs to B
7        add B to end of L
```

The resulting schedule has the desired properties by construction. To see that IncMerge can run in linear time, we need two observations. First, the speed of each block can be computed in constant time if prefix sums of the work are precomputed (in $O(n)$ time) and the amount of energy remaining is updated each time the list L is changed. Second, the loop in lines 4–6 takes $O(n)$ time total since each job ceases to be the first job of a block only once.

We also note that IncMerge has desirable numerical properties for the special case when power = speed$^\alpha$ for integral α, which includes systems obeying the cube-root rule ($\alpha = 3$). Specifically, the test in line 4 can be performed with rational arithmetic. The speed of a block other than the last is rational because its speed is its work over its duration (both integers). The energy used by one of these blocks is also rational since it is the product of the block's work and its speed to the $(\alpha - 1)$st power. This means that the energy of the last block is also rational since it is the energy budget (an integer) minus the sum of energies used by other blocks. Computing the speed of the last block would require taking a root of this energy, but that can be avoided for line 4 by instead comparing speed to the $(\alpha - 1)$st. (Computing a root is still necessary to find the achieved makespan since this depends on the actual speed of the last block.)

3.2 Finding all non-dominated schedules

A slight modification of IncMerge finds all non-dominated schedules. Intuitively, the modified algorithm enumerates all optimal configurations (i.e. ways to break the jobs into blocks) by starting with an "infinite" energy budget and gradually lowering it. To start this process, run IncMerge as above, but omit the merging step for the last job, essentially assuming the energy budget is large enough that the last job runs faster than its predecessor. To find each subsequent configuration change, calculate the energy budget at which the last two blocks merge. Until this value, only the last block changes speed. Thus, we can easily find the relationship between makespan and energy consumption for a single configuration and the curve of all non-dominated schedules is constructed by combining these. The curve for an instance with three jobs and power = speed^3 is plotted in Fig. 1. The configuration changes occur at energy 8 and 17, but they are not readily identifiable from the figure because

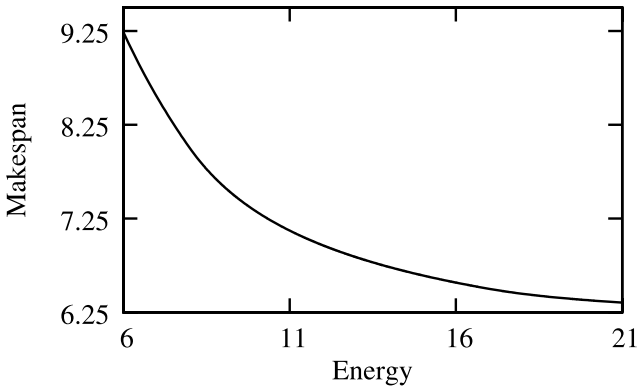

Fig. 1 Makespan as a function of energy in non-dominated schedules, for instance with $r_1 = 0$, $w_1 = 5$, $r_2 = 5$, $w_2 = 2$, $r_3 = 6$, $w_3 = 1$, and power = speed^3

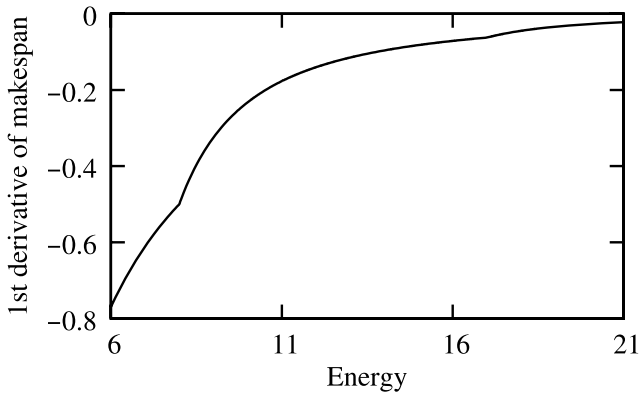

Fig. 2 First derivative of makespan as a function of energy in non-dominated schedules, for instance with $r_1 = 0$, $w_1 = 5$, $r_2 = 5$, $w_2 = 2$, $r_3 = 6$, $w_3 = 1$, and power = speed^3

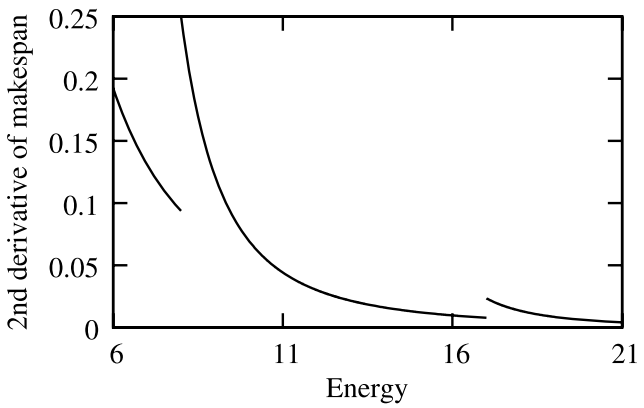

Fig. 3 Second derivative of makespan as a function of energy in non-dominated schedules, for instance with $r_1 = 0$, $w_1 = 5$, $r_2 = 5$, $w_2 = 2$, $r_3 = 6$, $w_3 = 1$, and power = speed^3

makespan is a continuous function of energy and its first derivative is also continuous. Higher derivatives are discontinuous at the configuration changes. Figures 2 and 3 show the first and second derivatives.

3.3 More general power functions

Now we show that algorithm IncMerge also works for more general power functions. We first extend it to power functions that are continuous and convex, but not strictly convex. Again, we begin by establishing the properties listed in Lemma 7.

Lemma 8 *If the power function is convex, there is an optimal solution with the properties listed in Lemma 7.*

Proof We begin with properties that follow trivially from our earlier discussion. We may assume jobs run in order of their release times (Property 1) since the proof of Lemma 2 does not assume anything about the power function. Each job can be made to run at a single speed (Property 2) by setting its speed to the average speed as in the proof of Lemma 3. Since the power function is convex, this does not increase energy consumption. Similarly, we can remove idle time (Property 3) by slowing jobs as in the proof of Lemma 4. Again, this does not increase energy consumption. In addition, any optimal solution runs its last block as quickly as possible (Property 6).

Establishing the other two properties is a bit more complicated. For each, we give a procedure to modify the schedule until the desired property holds while maintaining the existing properties. We use a variation of speed swapping that simultaneously speeds up one job and slows down another so that the makespan and total energy are unchanged. This can be achieved by using the previous type of speed swapping and then setting all work to run at its job's average speed.

We first make every job in a block run at the same speed (Property 4). Examine the jobs one at a time in execution order. Let J denote the job currently being examined. If J runs faster than the average speed for its block, speed-swap between it and one or more slower jobs in the block to bring its speed down to the average without raising the speed of the slower jobs above the average. This change does not start any job before its release since J lengthens by the total amount other jobs are shortened. If J runs slower than average, speed-swap with a job in the block that runs faster than average. Swap until either of the jobs runs at the average speed or some job between the swapping pair starts at its release time. (The second condition prevents the new schedule from violating release times.) If the faster job reaches the average speed, select another fast job to continue speed swapping. If the start time of some job reaches its release time, split the block at that point and recompute the average speed of each block. This process terminates because each pair of jobs is involved in a speed swap at most once between block splits and the number of block splits is at most $n - 1$.

Next, we adjust the schedule so that the block speeds are non-decreasing (Property 5). Whenever we have adjacent blocks where the first runs faster, we speed-swap, slowing every job in the earlier block and accelerating every job in the later block until all run at the same speed. This merges that pair of blocks. Again, the process terminates because each pair of jobs speed-swaps at most once between block merges and there can be at most $n-1$ merges. □

Now we are ready to show that IncMerge finds an optimal solution for all convex power functions.

Theorem 9 IncMerge *finds an optimal solution for all continuous convex power functions.*

Proof For brevity, let IncMerge denote the schedule output by algorithm IncMerge. Suppose to the contrary that OPT is an optimal schedule satisfying Lemma 2 with lower makespan than IncMerge. By Lemma 8, we may assume that OPT has the properties listed in Lemma 7. Then, by Lemma 7, IncMerge and OPT have the same blocks except that the jobs in IncMerge's last block form several blocks in OPT. Consider one of these blocks (i, j) that is not last in OPT. The jobs of block (i, j) run faster in OPT than in IncMerge since they finish before r_{j+1} in OPT, but not in IncMerge. Since IncMerge runs its last block as quickly as allowed by the available energy, OPT uses more energy for jobs $J_i, \ldots, J_j$ than IncMerge. Thus, OPT has less energy available for its last block than these same jobs use in IncMerge and must therefore run them no faster than IncMerge runs its last block. Since earlier block (i, j) ran faster than the last block of IncMerge, this contradicts the property that block speeds are non-decreasing. □

Our next extension is relatively minor, to convex power functions with a maximum possible speed. This does not affect the algorithm other than making it impossible to improve the makespan beyond the value achieved when the speed of the last block runs at its maximum speed.

Our final extension is to discrete power functions, those with only a finite number of possible speeds. To use IncMerge with such a power function, convert the power function into a piecewise linear power function by drawing segments between all points and taking the *lower hull* of the result, i.e. those points having the smallest power consumption for a given speed. The speed/power values on a segment are linear combinations of the endpoints and can be achieved by switching the processor speed between the values of the endpoints. The resulting power function is convex with a maximum possible speed, which we have already shown to be solvable by IncMerge.

4 Impossibility of exactly minimizing total flow time

We have completely solved uniprocessor power-aware makespan by showing how to compute all non-dominated schedules, forming a curve such as in Fig. 1. We have already observed that the analogous figure from previous work on total flow time was plotted with gaps where the optimal configuration involves one job completing exactly as another is released. We now show that these gaps cannot be filled exactly.

Theorem 10 *If power* $=$ *speed*^{3}, *there is no exact algorithm to minimize total flow time for a given energy budget using exact real arithmetic, including the extraction of roots, even on a uniprocessor with equal-work jobs.*

Proof We show that a particular instance cannot be solved exactly. Let jobs J_1 and J_2 arrive at time 0 and job J_3 arrive at time 1, each requiring one unit of work. We seek the minimum-flow schedule using 9 units of energy. Again, we use σ_i to denote the speed of job J_i. Thus,

$$\sigma_1^2 + \sigma_2^2 + \sigma_3^2 = 9. \tag{1}$$

For energy budgets between approximately 8.43 and approximately 11.54, the optimal solution finishes job J_2 at time 1. Therefore,

$$\frac{1}{\sigma_1} + \frac{1}{\sigma_2} = 1 \tag{2}$$

and Theorem 1 gives us that

$$\sigma_1^3 = \sigma_2^3 + \sigma_3^3. \tag{3}$$

Substituting (2) into (1) and (3), followed by algebraic manipulation gives

$$\begin{aligned} &2\sigma_2^{12} - 12\sigma_2^{11} + 6\sigma_2^{10} + 108\sigma_2^9 - 159\sigma_2^8 - 738\sigma_2^7 \\ &\quad + 2415\sigma_2^6 - 1026\sigma_2^5 - 5940\sigma_2^4 + 12150\sigma_2^3 \\ &\quad - 10449\sigma_2^2 + 4374\sigma_2 - 729 = 0. \end{aligned}$$

According to the GAP system (GAP Group 2006), the Galois group of this polynomial is not solvable. This implies the theorem by a standard result in Galois theory (cf. Dummit and Foote 1991, p. 542). □

This proof is based on an argument by Bajaj (1988) for an unrelated problem.

Since an arbitrarily-good approximation algorithm is known for total flow time, one interpretation of Theorem 10 is that exact solutions do not have a nice representation even allowing radicals. For most applications, the approximation is sufficient since finite precision is the normal state of affairs in computer science. Only an exact algorithm such as IncMerge can give closed-form solutions suitable for symbolic computation, however.

5 Multiprocessor scheduling

Now we consider power-aware scheduling on a multiprocessor where all the processors use a shared energy supply. Note that we restrict our attention to jobs that only run on a single processor (*serial jobs*). This corresponds to scheduling a computer with a multi-core processor or a server farm concerned only about total energy consumption and not the consumption of each machine separately. Except where explicitly stated otherwise, the results in this section assume that the power function is strictly convex. Recall that we assume jobs cannot migrate between processors during execution.

5.1 Distributing jobs to processors

We begin by showing how to assign equal-work jobs to processors for scheduling metrics having two properties. A metric is *symmetric* if it is not changed by permuting the job completion times. A metric is *non-decreasing* if it does not decrease when any job's completion time increases. Both makespan and total flow time have these properties, but some metrics do not. For example, total weighted flow time is not symmetric.

To prove our results, we need some notation. For schedule A and job J_i, let $\text{proc}^A(i)$ denote the index of the processor running job J_i and $\text{succ}^A(i)$ denote the index of the job run after J_i on processor $\text{proc}^A(i)$. Also, let $\text{after}^A(i)$ denote the portion of the schedule running on processor $\text{proc}^A(i)$ after the completion of job J_i, i.e. the jobs running after job J_i together with their start and completion times. We omit the superscript when the schedule is clear from context.

We begin by observing that job start times and completion times occur in the same order.

Lemma 11 *If* OPT *is an optimal schedule for equal-work jobs under a symmetric non-decreasing metric, then* $S_i^{\text{OPT}} < S_j^{\text{OPT}}$ *implies* $C_i^{\text{OPT}} \le C_j^{\text{OPT}}$.

Proof Suppose to the contrary that $S_i^{\text{OPT}} < S_j^{\text{OPT}}$ and $C_i^{\text{OPT}} > C_j^{\text{OPT}}$. Clearly, jobs J_i and J_j must run on different machines. We create a new schedule OPT′ from OPT. All jobs on machines other than $\text{proc}(i)$ and $\text{proc}(j)$ are scheduled exactly the same as are those that run before jobs J_i and J_j. We set the completion time of job J_i in OPT′ to C_j^{OPT} and the completion time of job J_j in OPT′ to C_i^{OPT}. We also switch the suffixes of jobs following these two, i.e. run $\text{after}(i)$ on processor $\text{proc}(j)$ and run $\text{after}(j)$ on processor $\text{proc}(i)$. Job J_i still has positive processing time since $S_i^{\text{OPT}'} = S_i^{\text{OPT}} < S_j^{\text{OPT}} < C_j^{\text{OPT}} = C_i^{\text{OPT}'}$. (The processing time of job J_j increases so it is also positive.) Thus, OPT′ is a valid schedule. The metric values for OPT and OPT′ are the same since this change only swaps the completion times of jobs J_i and J_j.

We complete the proof by showing that OPT′ uses less energy than OPT. Since the power function is strictly convex, it suffices to show that both jobs have longer processing time in OPT′ than job J_j did in OPT. Job J_j ends later so its processing time is clearly longer. Job J_i also has longer processing time since runs throughout the time OPT runs job J_j, but starts earlier. □

Using Lemma 11, we prove that an optimal solution exists with the jobs placed in *cyclic order*, i.e. job J_i runs on processor $(i \bmod m) + 1$.

Theorem 12 *There is an optimal schedule for equal-work jobs under any symmetric non-decreasing metric with the jobs placed in cyclic order.*

Proof We can place job J_1 on processor 1 without loss of generality. We complete the proof by giving a procedure to extend the prefix of jobs placed in cyclic order. Let OPT be an optimal schedule that places jobs $J_1, J_2, \ldots, J_{i-1}$ in cyclic order, but not job J_i. To simplify notation, we create dummy jobs $J_{-(m-1)}, J_{-(m-2)}, \ldots, J_0$, with job $J_{-(m-i)}$ assigned to processor i. By assumption, $\text{succ}(i-m) \neq i$. Let J_p be the job preceding job J_i, i.e. the job such that $\text{succ}(p) = i$. Since the first $i-1$ jobs are placed in cyclic order, if we assume (without loss of generality) that jobs starting at the same time finish in order of increasing index, then Lemma 11 implies that $C_{i-m}^{\text{OPT}} \le C_p^{\text{OPT}}$.

To complete the proof, we consider 3 cases. In each, we use OPT to create an optimal schedule assigning job J_i to processor $(i \mod m) + 1$, contradicting the definition of i.

Case 1. Suppose no job follows job J_{i-m}. We modify the schedule by moving $\text{after}(p)$ to follow J_{i-m} on processor $(i \mod m) + 1$. Since $C_{i-m}^{\text{OPT}} \le C_p^{\text{OPT}}$ and $\text{after}(p)$ was able to follow job J_p, it can also follow job J_{i-m}. The resulting schedule has the same metric value and uses the same energy, so it is also optimal.

Case 2. Suppose J_{i-m} is not the last job assigned to processor $\text{proc}(i-m)$ and $C_p^{\text{OPT}} < r_{\text{succ}(i-m)}$. We extend the cyclic order by swapping $\text{after}(p)$ and $\text{after}(i-m)$. This does not change the amount of energy used. To show that it gives a valid schedule, we need to show that jobs J_p and J_{i-m} complete before $\text{after}(i-m)$ and $\text{after}(p)$. Job J_p ends by time $S_{\text{succ}(i-m)}^{\text{OPT}}$ by the assumption that $C_p^{\text{OPT}} < r_{\text{succ}(i-m)}$. Job J_{i-m} ends by time $S_{\text{succ}(p)}^{\text{OPT}}$ since $C_{i-m}^{\text{OPT}} \le C_p^{\text{OPT}}$.

Case 3. Suppose J_{i-m} is not the last job assigned to processor $\text{proc}(i-m)$ and $C_p^{\text{OPT}} \ge r_{\text{succ}(i-m)}$. In this case, we swap the jobs $J_{\text{succ}(i-m)}$ and $J_{\text{succ}(p)} = J_i$, but leave the schedules the same. In other words, we run job $J_{\text{succ}(i-m)}$ from time S_i^{OPT} to time C_i^{OPT} on processor $\text{proc}(p)$ and we run job J_i from time $S_{\text{succ}(k)}^{\text{OPT}}$ to time $C_{\text{succ}(k)}^{\text{OPT}}$ on processor $\text{proc}(k)$. The schedules have the same metric value and

each uses the same amount of energy. To show that we have created a valid schedule, we need to show that jobs $J_{\text{succ}(i-m)}$ and J_i are each released by the start time of the other. Job $J_{\text{succ}(i-m)}$ was released by time S_i^{OPT} since $C_p^{\text{OPT}} \geq r_{\text{succ}(i-m)}$. Job J_i was released by time $S_{\text{succ}(i-m)}^{\text{OPT}}$ since $r_i \leq r_{\text{succ}(i-m)}$. (Recall that a job with index greater than i follows job J_{i-m}, so $r_i \leq r_{\text{succ}(i-m)}$.) □

A simpler proof suffices if we specify the makespan metric since then an optimal schedule has no idle time. Thus, $r_{\text{succ}(i-m)} \leq C_{i-m}^{\text{OPT}} \leq C_l^{\text{OPT}}$ and case 2 is eliminated.

Note that Albers et al. (2007) use a similar idea (discovered independently) for scheduling jobs with deadlines. Specifically, they show that placing equal-work jobs in cyclic order allows deadline feasibility with minimum energy when the deadlines have the property that $r_i < r_j$ implies that the deadline of job J_i is no later than the deadline of job J_j for all i and j. Their result is not implied by ours since deadline feasibility is not symmetric, but the proof has a similar flavor.

5.2 Multiprocessor algorithms

Once the jobs are assigned to processors, we can use slight modifications of IncMerge and the total flow time algorithm of Pruhs et al. (2004). In order to do this, we make the following simple observations that relate the schedules on each processor:

1. For makespan, each processor must finish its last job at the same time or slowing the processors that finish early would save energy.
2. For total flow time, each processor runs its last job at the same speed or running them at the average speed would save energy.

For makespan, the resulting algorithm begins by running IncMerge separately on each processor up to the point of determining the speed of the last block. These last blocks must end at the same time and exactly consume the remaining energy. Next, the algorithm determines the completion time of these blocks (i.e. the makespan); how, we describe below. From this, the speed of each last block is calculated. If the non-decreasing speed property is violated on any processor, the last two blocks on that processor are merged as in IncMerge and the makespan is recomputed. Once no change is needed, the resulting makespan is returned.

It remains to explain how the algorithm determines the makespan for a given block structure. For each last block, we have a starting time and an amount of work. We need to solve for the time t at which these blocks end. The speed of each last block has the form

$$\frac{\text{work}}{t - \text{start\_time}}. \tag{4}$$

The energy of each last block is computed from these speeds. The resulting values are summed and set equal to the remaining energy. The tricky part is then to solve this equation for t, but a binary search style algorithm can find an arbitrarily-good approximation in a number of steps logarithmic in the reciprocal of the desired accuracy (expressed as a percent error). Let L denote this reciprocal; then the energy used with a particular value of t is evaluated $O(\log L)$ times for each candidate block structure.

The exact runtime of this algorithm depends on how fast a job's energy consumption can be computed from its speed. For power = speed$^\alpha$, solving for the energy used with a particular value of t takes $O(m^2)$ time since the equation gets a term of the form shown in (4) from each processor; multiplying to get rid of the denominators creates 1 term with m factors and m terms with $m-1$ factors. Finding these roots takes $O(nm^2 \log L)$ since there can be n changes to the block structure. This aspect of the algorithm takes the longest, so its overall runtime is also $O(nm^2 \log L)$.

Although an exact algorithm would be preferable to an arbitrarily-good approximation, the multiprocessor makespan cannot be exactly solved in general, even with exact real arithmetic and the ability to take roots. Consider an instance on five processors consisting of jobs J_1–J_5 where $r_i = i - 1$ and each job requires one unit of work. Let power = speed$^\alpha$. Following the algorithm above, each job gets it own processor and the energy used is

$$\sum_{i=1}^{5} \frac{1}{t-i+1}.$$

If the energy budget is 1, solving for the makespan is equivalent to finding a root of

$$t^{10} - 20t^9 + 165t^8 - 720t^7 + 1733t^6 - 1980t^5 - 255t^4 + 3640t^3 - 4424t^2 + 2400t - 576 = 0.$$

As in the proof of Theorem 10, GAP reports that the Galois group of this polynomial is not solvable so there is no exact algorithm for the multiprocessor makespan.

Next we give our multiprocessor extension of the Pruhs et al. (2004) algorithm for total flow when power = speed$^\alpha$. The uniprocessor algorithm is based on finding whether each job should finish before, after, or exactly at the release time of its successor. We call an assignment of these relationships to each job a *configuration* of the jobs. For a given configuration, Theorem 1 relates the speed of each job to the speed of the last job and allows us to find the flow to within an arbitrarily-small error. To find the correct configuration, the algorithm starts by assuming an sufficiently-large energy budget so that each job finishes before its successor's arrival. Then, the algorithm gradually lowers the speed of the last job, detecting each configuration change as it occurs. This

algorithm takes $O(n^2 \log L)$ time, where L is the reciprocal of the desired accuracy. For a multiprocessor problem, the job energies on a single processor are related as in the uniprocessor setting. Since the last job on each processor runs at the same speed, the jobs speeds are actually related between processors as well. In fact, the equations relating the speed of each job are identical to those that would occur if every processor's jobs ran on a single processor with idle periods separating the jobs assigned to different processors. Since assigning the jobs to processors takes only linear time, the multiprocessor algorithm takes the same $O(n^2 \log L)$ time as the uniprocessor version.

5.3 Hardness for jobs requiring different amount of work

Theorem 12 allows us to solve multiprocessor makespan for equal-work jobs. Unfortunately, the general problem is NP-hard.

Theorem 13 *The laptop problem is NP-hard for nonpreemptive multiprocessor makespan, even if all jobs arrive at the same time.*

As stated previously, this result was proven by Shabtay and Kaspi (2006) when the time to complete job J_i is $(w_i/x_i)^k$, where k is a constant and x_i is the amount of resource devoted to job J_i, with $\sum x_i$ bounded. Our proof is essentially the same as theirs; we include it for completeness.

Proof of Theorem 13 We give a reduction from the NP-complete problem PARTITION (Garey and Johnson 1979):

> PARTITION: Given a multiset $A = \{a_1, a_2, \ldots, a_n\}$, does there exist a partition of A into A_1 and A_2 such that $\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i$?

Let $B = \sum_{i=1}^{n} a_i$. We assume B is even since otherwise no partition exists. We create a scheduling problem from an instance of PARTITION by creating a job J_i for each a_i with $r_i = 0$ and $w_i = a_i$. Then we ask whether a 2-processor schedule exists with makespan $B/2$ and a power budget allowing work B to run at speed 1.

From a partition, we can create a schedule where each processor runs the jobs corresponding to one of the A_i at speed 1. For the other direction, the convexity of the power function implies that all jobs run at speed 1 so the work must be partitioned between the processors. □

Pruhs et al. (2005) observed that the special case of all jobs arriving together has a PTAS based on a load balancing algorithm of Alon et al. (1997) that approximately minimizes the L_α norm of loads.

6 Discussion

The study of power-aware scheduling algorithms began only recently so there are many possible directions for future work. We are particularly interested in online algorithms. Our results on the structure of optimal solutions may help with this task, but the problem seems quite difficult. If the algorithm cannot know when the last job has arrived, it must balance the need to run quickly to minimize makespan if no other jobs arrive against the need to conserve energy in case more jobs do arrive. One solution is to follow the direction of Albers and Fujiwara (2006) and Bansal et al. (2007), minimizing makespan plus energy consumption rather than makespan alone.

Another open problem is how to handle jobs with different work requirements while minimizing multiprocessor makespan. Theorem 13 shows that finding an exact solution is NP-hard, but this does not rule out the existence of a high-quality approximation algorithm. As mentioned above, there is a PTAS based on load balancing when all jobs arrive together. It would be interesting to see if the ideas behind this PTAS can be adapted to take release times into consideration. Alternately, it would be interesting to show that the problem is strongly NP-hard.

We would also like to find an approximation for uniprocessor power-aware scheduling to minimize total flow time when the jobs have different work requirements. It is not hard to show that the relationships of Theorem 1 hold even in this case. Preemptions can also be incorporated with the preempted job taking the role of job J_{i+1} in the second relationship of that theorem. Thus, the problem reduces to finding the optimal configuration.

We would also like to see theoretical research using models that more closely resemble real systems. The most obvious change is to use discrete power functions, which we did in this paper. Imposing minimum and/or maximum speeds is one way to partially incorporate this aspect of real systems without going all the way to the discrete case. Another feature of real systems is that slowing down the processor has less effect on memory-bound sections of code since part of the running time is caused by memory latency. There is already some simulation-based work attempting to exploit this phenomenon (Xie et al. 2003). Finally, real systems incur overhead to switch speeds because the processor must stop while the voltage is changing. This overhead is fairly small, but discourages algorithms requiring frequent speed changes.

Acknowledgements We thank Jeff Erickson for introducing us to the work of Bajaj (Bajaj 1988) on using Galois theory to prove hardness results. We also benefited from comments made by the anonymous referees and from discussions with Erin Chambers, Dan Cranston, Sariel Har-Peled, Steuard Jensen, and Andrew Leahy.

References

Advanced Micro Devices. (2004). AMD Athlon 64 processor power and thermal data sheet (version 3.43), October 2004. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/30430.pdf.

Albers, S., & Fujiwara, H. (2006). Energy-efficient algorithms for flow time minimization. In *Proceedings of the 23rd international symposium on theoretical aspects of computer science* (pp. 621–633).

Albers, S., Müller, F., & Schmelzer, S. (2007). Speed scaling on parallel processors. In *Proceedings of the 19th annual ACM symposium on parallelism in algorithms and architectures* (pp. 289–298).

Alon, N., Azar, Y., Woeginger, G. J., & Yadid, T. (1997). Approximation schemes for scheduling. In *Proceedings of the 8th annual ACM-SIAM symposium on discrete algorithms* (pp. 493–500).

Bajaj, C. (1988). The algebraic degree of geometric optimization problems. *Discrete Comput. Geom.*, *3*, 177–191.

Bansal, N., Pruhs, K., & Stein, C. (2007). Speed scaling for weighted flow time. In *Proceedings of the 18th annual ACM-SIAM symposium on discrete algorithms* (pp. 805–813).

Brooks, D. M., Bose, P., Schuster, S. E., Jacobson, H., Kudva, P. N., Buyuktosunoglu, A., Wellman, J.-D., Zyuban, V., Gupta, M., & Cook, P. W. (2000). Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, *20*(6), 26–44.

Bunde, D. P. (2006). Power-aware scheduling for makespan and flow. In *Proceedings of the 18th annual ACM symposium on parallelism in algorithms and architectures* (pp. 190–196).

Chekuri, C., & Bender, M. A. (2001). An efficient approximation algorithm for minimizing makespan on uniformly related machines. *Journal of Algorithms*, *41*, 212–224.

Chen, J.-J., Kuo, T.-W., & Lu, H.-I. (2005). Power-saving scheduling for weakly dynamic voltage scaling devices. In *Lecture notes in computer science: Vol. 3608. Proceedings of the 9th workshop on algorithms and data structures* (pp. 338–349). Berlin: Springer.

Chudak, F. A., & Shmoys, D. B. (1997). Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. In *Proceedings of the 8th annual ACM-SIAM symposium on discrete algorithms* (pp. 581–590).

Dummit, D. S., & Foote, R. M. (1991). *Abstract algebra*. Englewood Cliffs: Prentice-Hall.

El Gamal, A., Nair, C., Prabhakar, B., Uysal-Biyikoglu, E., & Zahedi, S. (2002). Energy-efficient scheduling of packet transmissions over wireless networks. In *Proceedings of the IEEE INFOCOM* (pp. 1773–1782).

GAP Group. (2006). GAP system for computational discrete algebra. http://turnbull.mcs.st-and.ac.uk/~gap/ (viewed January 2006).

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. New York: Freeman.

Irani, S., & Pruhs, K. R. (2005). Algorithmic problems in power management. *SIGACT News*, *32*(2), 63–76.

Keslassy, I., Kodialam, M., & Lakshman, T. V. (2003). Faster algorithms for minimum-energy scheduling of wireless data transmissions. In *Proceedings of the modeling and optimization in mobile, ad hoc and wireless networks*.

Mudge, T. (2001). Power: A first-class architectural design constraint. *Computer*, *34*(4), 52–58.

Pinedo, M. L. (2005). *Planning and scheduling in manufacturing and services*. *Springer series in operations research*. New York: Springer.

Pruhs, K., Uthaisombut, P., & Woeginger, G. (2004). Getting the best response for your erg. In *Lecture notes in computer science: Vol. 3111. Proceedings of the 9th Scandinavian workshop on algorithm theory* (pp. 14–25). Berlin: Springer.

Pruhs, K., van Stee, R., & Uthaisombut, P. (2005). Speed scaling of tasks with precedence constraints. In *Lecture notes in computer science: Vol. 3879. Proceedings of the 3rd workshop on approximation and online algorithms* (pp. 307–319). Berlin: Springer.

Rudin, W. (1987). *Real and complex analysis* (3rd ed.) New York: McGraw-Hill.

Shabtay, D., & Kaspi, M. (2006). Parallel machine scheduling with a convex resource consumption function. *European Journal of Operational Research*, *173*, 92–107.

Shabtay, D., & Steiner, G. (2007). A survey of scheduling with controllable processing times. *Discrete Applied Mathematics*, *155*, 1643–1666.

Shakhlevich, N. V., & Strusevich, V. A. (2005). Pre-emptive scheduling problems with controllable processing times. *Journal of Scheduling*, *8*(3), 233–253.

Shakhlevich, N. V., & Strusevich, V. A. (2006). Single machine scheduling with controllable release and processing parameters. *Discrete Applied Mathematics*, *154*, 2178–2199.

Tiwari, V., Singh, D., Rajgopal, S., Mehta, G., Patel, R., & Baez, F. (1998). Reducing power in high-performance microprocessors. In *Proceedings of the 35th ACM/IEEE design automation conference* (pp. 732–737).

Uysal-Biyikoglu, E., & El Gamal, A. (2004). On adaptive transmission for energy efficiency in wireless data networks. *IEEE Transactions on Information Theory*, *50*(12), 3081–3094.

Uysal-Biyikoglu, E., Prabhakar, B., & El Gamal, A. (2002). Energy-efficient packet transmission over a wireless link. *IEEE/ACM Transactions on Networking*, *10*(4), 487–499.

Weiser, M., Welch, B., Demers, A., & Shenker, S. (1994). Scheduling for reduced CPU energy. In *Proceedings of the 1st symposium on operating systems design and implementation* (pp. 13–23).

Xie, F., Martonosi, M., & Malik, S. (2003). Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the 2003 ACM SIGPLAN conference on programming language design and implementation* (pp. 49–62).

Yao, F., Demers, A., & Shenker, S. (1995). A scheduling model for reduced CPU energy. In *Proceedings of the 36th symposium on foundations of computer science* (pp. 374–382).