# Scheduling Heterogeneous Processors Isn't As Easy As You Think

Anupam Gupta* Sungjin Im † Ravishankar Krishnaswamy‡ Benjamin Moseley §
Kirk Pruhs¶

*"With multi-core it's like we are throwing this Hail Mary pass down the field and now we have to run down there as fast as we can to see if we can catch it."*
— David Patterson, UC Berkeley computer science professor

## Abstract

We consider preemptive online scheduling algorithms to minimize the total weighted/unweighted flow time plus energy for speed-scalable heterogeneous multiprocessors. We show that the well-known priority scheduling algorithms *Highest Density First*, *Weighted Shortest Elapsed Time First*, and *Weighted Late Arrival Processor Sharing*, are not $O(1)$-speed $O(1)$-competitive for the objective of weighted flow even in the special case of fixed variable speed processors (aka the related machines setting). This illustrates that scheduling heterogeneous multiprocessors is a different, and algorithmically more challenging problem, than scheduling homogeneous multiprocessors.

We then show that a variation of the non-clairvoyant algorithm *Late Arrival Processor Sharing* coupled with a non-obvious speed scaling algorithm is scalable for the objective of unweighted flow plus energy on speed-scalable multiprocessors. This is the first provably scalable *non-clairvoyant* algorithm on heterogeneous multiprocessors, even in the related machines setting, for the objective of total (unweighted) flow time.

## 1 Introduction

Around 2002, the consequences of Moore's law finally impacted computer processor designers as they hit a "thermal wall", where it was no longer economically viable to cool the ever-hotter traditional uniprocessor architectures. One technology adopted to surmount this thermal wall is multiprocessor chips. The common rule of thumb is that the power used by a processor is roughly cubic in the speed of the processor. In theory $m$ processors with speed $s/m$ could potentially handle the same load as one speed-$s$ processor, but with a factor of $1/m^2$ less power. Moore's gap, which is the difference in the achievable performance predicted by Moore's law and the actual performance of commercial processors, is largely explained by difficulty of getting many slower processors to approximate the performance of one fast processor in practice.

Current commercial chip architectures mostly commonly consist of a homogeneous collection of identical processors. However, many computer architects believe that architectures consisting of *heterogeneous* processors/cores will be the dominant architectural design in the future [7, 18, 19, 20, 21]. The main advantage of a heterogeneous architecture over a homogeneous architecture is that it allows for the inclusion of processors whose design is specialized for particular types of jobs, with the intent that jobs be assigned to a processor best suited for that job. Most notably, it is envisioned that these heterogeneous architectures will consist of a small number of high-power high-performance processors for critical jobs, and a larger number of lower-power lower-performance processors for less critical jobs. Naturally, the lower-power processors would be more energy efficient in terms of the computation performed per unit of energy expended, and would generate less heat per unit of computation. An example of a such a heterogeneous multiprocessor is the STI Cell chip. For a given area and power budget, heterogeneous multiprocessor architectures can give a order of magnitude better performance than homogeneous multiprocessor architectures [15]. This makes research into scheduling policies for heterogeneous processors of fundamental importance (see the position paper [7] for further arguments about the importance of this research direction).

*Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA. anupamg@cs.cmu.edu. Supported in part by NSF awards CCF-0964474 and CCF-1016799.

†Department of Computer Science, University of Illinois, 201 N. Goodwin Ave., Urbana, IL 61801. im3@illinois.edu. Partially supported by NSF grants CCF-1016684.

‡Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA. ravishan@cs.cmu.edu . Supported in part by NSF awards CCF-0964474 and CCF-1016799, and an IBM Graduate Student fellowship.

§Department of Computer Science, University of Illinois, 201 N. Goodwin Ave., Urbana, IL 61801. bmosele2@illinois.edu. Partially supported by CCF-1016684.

¶Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260, USA. kirk@cs.pitt.edu. Supported in part by NSF grants CCF-0830558 and CCF-1115575, and an IBM Faculty Award.

Currently the most pervasive technology for achieving power heterogeneity is that of *speed-scalable* processors. Speed-scalable processors have a collection of available speeds, and the power consumed at the various speeds is a convex function of the speed. The speed of the processors can be dynamically scaled over time. A system that sits atop a speed-scalable processor needs not only an online scheduling policy to determine which job to run on which processor, but also a speed scaling policy for setting the speed of these processors. In the homogeneous setting, each processor runs at the same speed when using a particular power setting while in the heterogeneous setting the speed for a given power depends on the specific processor being considered.

Following the line of research in [14, 13], we investigate worst-case performance guarantees (or competitive ratios) achievable by algorithms for scheduling heterogeneous multiprocessor architectures. Throughout the paper we focus on a type of heterogeneous multiprocessor scheduling which is best described as related heterogeneous multiprocessors and is defined as follows. We adopt the following formal model as in [14]. We are given a collection of $m$ processors/machines, with processor $i$ having a *speed function* $Q_i$: for every value $P$, $Q_i(P)$ is the speed obtained when the processor is run at a power of $P$. Notice that the speed depends on the processor being considered. One can assume without loss of generality that $Q_i$ is concave, continuous, and $Q_i(0) = 0$ [2]. If processors are not running a job then they can be shut down, and consume no power. Jobs arrive in an online fashion over time, with job $J_j$ arriving in the system at its release/arrival time $r_j$. The job has a positive *size* $p_j$, and a positive *importance/weight* $w_j$. Each job can be scheduled on only one processor at each time and can be preempted. The goal is to devise a scheduling policy and an associated speed scaling policy to minimize some weighted combination of the average (weighted) flow time $\sum_j w_j F_j$ of the jobs and the total energy consumed. Here, the flow time $F_j$ of a job $J_j$ is the difference between its completion time and its release time. We also note that an important special case of this model is the *related machines* model, where each processor $i$ can only run at a single fixed speed $s_i$ and each processor consumes no power.

Before stating our findings, we review the current state of research along this line (readers unfamiliar with standard scheduling terminology are advised to consult Section 1.3).

**1.1 Current State of Affairs** We consider the popular and well-studied problems of minimizing the total weighted and unweighted flow time plus energy. We say that a processor runs at a *fixed speed* if there is only one possible speed for the processor and it consumes no energy. The current state of affairs (see Tables 1, 2 and 3 in the Appendix) can roughly be summarized as follows. For a single processor of fixed speed, the well-known priority algorithms[1] covered in standard introductory operating systems texts are known to be scalable (i.e. possess a constant competitive ratio when provided a processor that is a factor $(1 + \epsilon)$ faster than the optimal solution) for the unweighted case—these include SRPT (*shortest remaining processing time*), SJF (*shortest job first*), SETF (*shortest elapsed time first*), and their weighted versions are known to be scalable for the weighted case [17, 6, 11, 4, 3]. (See Appendix A for definitions of these algorithms.) Likewise, for a single processor that is speed-scalable, we can obtain near-optimal algorithms in the weighted or unweighted settings by combining the standard priority scheduling algorithms with a natural speed-scaling policy where the power is set to be a small multiple of the total weight of the unsatisfied jobs [5, 2, 1, 9, 10]. It is easy to see that such a speed-scaling policy is natural (for the objective of weighted flow times plus energy) because it balances the increase in the weighted flow time objective with the increase in the energy objective. Furthermore, many of these standard priority scheduling algorithms are known to be scalable for the problem of homogeneous fixed-speed *multi*processors [25, 12, 11].

For heterogeneous multiprocessors however, the landscape is not so well-charted. Scalable clairvoyant algorithms are known for weighted flow on fixed speed processors [8], and for weighted flow plus energy on speed-scalable processors [14]. These algorithms are quite different, and more complicated than the standard priority algorithms. It is also known that the non-clairvoyant scheduling algorithm *Processor Sharing* (or *Equipartition/Round Robin*) is $(2 + \epsilon)$-speed $O(1)$-competitive for the objective of unweighted flow plus energy on speed-scalable processors [13].

**1.2 Our Contributions** Based on conversations with our colleagues, the near universal expectation/intuition of scheduling researchers was that scheduling (related) heterogeneous multiprocessors should be similar to scheduling homogeneous processors. In particular, it was believed that the standard priority algorithms that are known to be scalable for the uniprocessor and for homogeneous multiprocessor problems, should also extend to the heterogeneous multiprocessor problem. The first contribution of this paper is to show that this intuition is not correct.

Most of the analysis techniques for scheduling algorithms in the homogeneous multiprocessor fixed speed setting do not extend to the heterogeneous fixed speed

[1]We say a scheduling algorithm is a *priority algorithm* if the jobs are assigned a single parameter (which can possibly change with time) called its priority, and the scheduling decision is based solely on each job's priority. For example, in SRPT, the priority of each job is simply the remaining processing time of the job.

multiprocessor setting for one or both of the following reasons. Firstly, contrary to conventional intuition, priority algorithms such as SRPT and SJF are not locally competitive[2] (even with any constant factor speed augmentation) as they are on a homogeneous fixed speed multiprocessor [23]. (See Appendix B.) Secondly, unlike the homogeneous case, it is difficult to establish lower bounds on when the optimal solution completes these jobs. E.g., the total work of a set of jobs divided by the total speed of the processors is not useful: even though a set of processors may have large aggregate speed, each individual processor may be very slow. For the same reason, the number of processors is not a useful quantity.

If we were to consider weights, the situation becomes more challenging. We show in Section 2 that the standard extension of priority algorithms for the *weighted* flow objective, namely HDF, WSETF, and WLAPS (*weighted latest arrival processor sharing*), are all not $O(1)$-speed $O(1)$-competitive, even for the related machines setting when machines have different but fixed speeds and consume no energy. Note that as mentioned above, these algorithms *are scalable* for the homogeneous case when all processors have the same speed [11, 12]. Intuitively, perhaps the underlying reason is that when we have both related machines and weighted jobs there is an extra dimension to the problem over both the cases of weighted jobs on homogeneous machines and unweighted jobs on heterogeneous machines. The natural extensions of priority algorithms fail to capture the interplay between these dimensions. We believe that this justifies the analysis of non-standard algorithms in [8, 14].

One natural question that arises from these negative results, and prior positive results, is whether any non-clairvoyant algorithm can be scalable on a heterogeneous multiprocessor system. Studying the performance of non-clairvoyant algorithms is of particular importance because schedulers in general purpose systems generally do not know the size of the job upon its arrival. In Section 3, we show that for *unweighted* flow plus energy, the answer is yes. That is, we show that the following algorithm, which combines the LAPS [11] (latest arrival processor sharing) policy with a non-intuitive speed-scaling policy is scalable for the objective of total flow plus energy on a heterogeneous multiprocessor. This improves upon the result of [13] which shows that the scheduling algorithm Equipartition is $(2 + \epsilon)$-speed $O(1)$-competitive for the same objective.

At a high level, the main technical difficulty in showing LAPS is scalable is the following. Consider the related machines model where machines have a fixed speed and do not consume any energy. In this case, typical arguments for LAPS on homogeneous multiprocessors proceed by (i) showing that we can treat $m$ identical machines as just a single processor of speed $m$ as long as we restrict each job to not run at more than unit speed at any time, and (ii) on this one machine instance, showing that we can just distribute the speed of the system among the $\epsilon n$ most recently arriving jobs, and still make enough progress on the overall objective. However, we run into trouble in both steps for heterogeneous systems. For (i) it is not clear what the single machine instance should be, because the machines could have vastly different speed profiles, and we can't therefore place such natural restrictions on jobs to capture the fact that a job can run only on one machine. So sticking with multiprocessors, the problem then with (ii) is that $\epsilon n$ could always be smaller than $m$, the number of machines. In this case it is not possible to fully utilize the resources of $m$ machines without running a job simultaneously on two machines, which is an infeasible schedule. However, we show that the algorithm which shares the $\epsilon n$ fastest machines between the $\epsilon n$ latest arriving jobs is scalable. We use this as a starting point for our general algorithm in the speed-scaling case. Because of the issues discussed above, our analysis is also forced to reason directly about a heterogeneous multiprocessor system. See Section 3 for more details.

We note that this is the *first* example of a scalable non-clairvoyant algorithm for speed-scalable heterogeneous processors, or even fixed-speed related machines. Moreover, due to strong lower bounds without resource augmentation [22], this is essentially the best positive result that we can hope for.

On the whole, our belief is that this paper suggests that scheduling heterogeneous multiprocessors may be inherently more difficult than scheduling homogeneous multiprocessors, or at the very least, require substantially different algorithms.

## 1.3 Review of Standard Scheduling Terminology

The flow $F_j$ of a job $J_j$ is its completion time $C_j$ minus its release time $r_j$. This is the amount of time that the job waits to be satisfied. The weighted flow for a job $J_j$ is $w_j F_j$, and the weighted flow for a schedule is $\sum_j w_j F_j$. A scheduler minimizing the weighted flow time focuses on minimizing the average weighted quality of service. The goal of the scheduler is to minimize the total weighted flow time plus the total energy used. The intuitive rationale for the objective of weighted flow plus energy can be understood as follows: assume that the possibility exists to invest $E$ units of energy to decrease the flow of jobs $J_1, \ldots, J_k$ by $x_1, \ldots, x_k$ respectively for some $k > 0$. An optimal scheduler for this objective would make such

[2]An algorithm is *locally competitive* if at all times the increase in the algorithm's objective is within a constant factor of the increase in the optimal solution's objective. For weighted flow time this means that the total weight of unsatisfied jobs in the algorithm's schedule is within a constant factor of the total weight of unsatisfied jobs in the optimal solution's schedule at all times.

an investment if and only if $\sum_{i=1}^{k} w_i x_i \geq E$. So the importance $w_j$ of job $J_j$ can be viewed as specifying an upper bound on the amount of energy that the system is allowed to invest to reduce $J_j$'s flow time by one unit of time. Hence jobs with higher weight are more important, since higher investments of energy are permissible to justify a fixed reduction in the job's flow time. One can consider many variations on this problem depending on the following factors:

- **Speed-Scalable Processors vs. Fixed-Speed Processors:** A fixed speed processor has only one allowable speed, the power used can be assumed to be zero without loss of generality. A speed-scalable processor can change its speed over time and the energy consumed depends on the speed used.
- **Homogeneous Multiprocessor vs. Heterogeneous Multiprocessor:** In the homogeneous setting, the speed function of every processor is the same. That is, each processor runs at the same speed for a given amount of power. However, at any given time, the processors can run at different speeds by using different powers. In the heterogeneous setting, each processor has its own specified speed function.
- **Unweighted vs. Weighted Jobs:** In the unweighted setting each job is of equal importance, i.e., all weights are assumed to be one. In the weighted case, jobs have varying importance/weights associated.
- **Clairvoyant vs. Non-Clairvoyant Scheduler:** A clairvoyant scheduler learns the job size when it is released. A non-clairvoyant scheduler does not learn a job's size and must make scheduling decisions without this information.

The most commonly used benchmark for online scheduling problems with strong lower bounds is the optimal offline schedule for the given instance but on slightly slower processors [17]. For $s, c \geq 1$, an algorithm $A$ is called *$s$-speed $c$-competitive* for the uniprocessor fixed-speed setting if $A$ with an $s$-speed processor is guaranteed to produce a schedule with objective value at most $c$ times the optimum on a unit speed processor. This generalizes to the multiprocessor speed-scaling setting by assuming that the speed processor $i$ runs at with power $P$ is $s$ times the speed the optimal algorithm can run processor $i$ at with power $P$. Intuitively, $s$-speed $O(1)$-competitive algorithm should be able to handle a load of $\frac{1}{s}$ of the server capacity [24]. The ultimate goal is to find an algorithm $A$ such that for any $\epsilon > 0$, a speed augmentation of $(1 + \epsilon)$ is enough for algorithm $A$ to achieve $O(1)$-competitiveness. Such an algorithm is called *scalable*.

## 2 Lower Bounds on Weighted Flow time on Related Machines

In this section we show that the standard priority algorithms for the weighted flow objective, namely HDF, WSETF, and the most natural adaptation of WLAPS (*weighted latest arrival processor sharing*), are not $O(1)$-speed $O(1)$-competitive for total weighted flow time on uniformly related machines. This is the heterogeneous processor setting where each machine runs at a fixed speed and consumes no power. When each machine consumes no power, the objective just simplifies to minimizing the total weighted flow time. As previously stated, this is a special case of the speed scaling heterogeneous processor setting with the objective of total weighted flow time plus energy. In each of the following subsections, we first explain how these priority algorithms generalize to heterogeneous machines, and then provide the lower bound examples.

### 2.1 Lower Bound for Highest Density First (HDF)

In HDF, the priority of a job is its density, i.e., job $J_i$ has a priority equal to its weight divided by its size $\frac{w_i}{p_i}$. The algorithm on a single processor, always schedules the highest density job. This naturally extends to related machines by scheduling the job of the $k^{th}$ highest density on the $k^{th}$ fastest machine at all times. The work of [6] shows that this algorithm is $O(1)$-speed $O(1)$-competitive when all machines have the same speed. However, when the speeds can be different, the following example shows that HDF has unbounded competitive ratio on related machines, even when provided any constant speed augmentation.

THEOREM 2.1. *For any constants $\alpha, B > 0$, there exists an instance $\mathcal{I}(\alpha, B)$ of related machines scheduling for which* HDF *is not $B$-competitive for the objective of weighted flow with a speed augmentation of $\alpha$.*

*Proof.* For this proof, let $\text{cost}(A)$ denote the total weighted flow time of an algorithm $A$. The instance $\mathcal{I}$ is defined in the following manner. There is a "fast" machine of speed $S$, and infinitely many "slow" machines of speed 1. At time $t = 0$, a "heavy" job of weight $W$ and length $L$ arrives. Then, at each time $\frac{i}{\alpha S}$, for integer $0 \leq i < SL$, a "small" job of weight $w = 4W/L$ and length 1 arrives (all the parameters $S, W, L$ will be set appropriately when required).

Note that each small job has a density $w$, which is greater than the density of the heavy job, $W/L$. Hence as long as there is a small job that is unfinished, the heavy job will not run on the fast machine in HDF's schedule. Now, since each small job completes on the fast machine after a time of $\frac{1}{\alpha S}$, the next small job arrives as soon as its preceding small job is finished by HDF by the way we have set up the instance. This is repeated until all small jobs complete and takes exactly $SL\frac{1}{\alpha S} = \frac{L}{\alpha}$ units of time. This implies that the heavy job is processed entirely on slow machines by HDF, and as a result, HDF incurs a weighted flow time of at least $\text{cost}(\text{HDF}) \geq W\frac{L}{\alpha}$ .

The optimal solution, however, will run the heavy job on the fast machine until completion, and run each small job on a *dedicated* unit speed machine. Recall that there are enough slow machines to run all small jobs simultaneously. The cost of the optimal solution is $\text{cost}(\mathsf{Opt}) = W\frac{L}{S} + wSL = W\frac{L}{S} + 4WS$. We set the length of the heavy job so that $W\frac{L}{S} = 4SW$, i.e., $L = 4S^2$. This implies a lower bound on the competitive ratio of $W\frac{L}{\alpha}\Big/2W\frac{L}{S} = \frac{S}{2\alpha}$. To complete the proof, we set $S = 4\alpha B$.

## 2.2 A Lower Bound for Weighted Shortest Elapsed Time First (WSETF)

In this section we show a lower bound on the well-known algorithm WSETF (*weighted shortest elapsed first*) in the related machines setting for total weighted flow time. We begin by describing the algorithm: at any time $t$, let $q_j(t)$ denote the the amount of work that job $J_j$ has been processed by. For any unfinished job $J_j$, define its priority at time $t$ to be $w_j/q_j(t)$. Then WSETF assigns the job with the $i^{th}$ highest priority on the $i^{th}$ fastest machine. We remark that this algorithm is scalable on a single processor [4]. We now show that an instance quite similar to the bad example for HDF is also bad for WSETF.

THEOREM 2.2. *For any constants $\alpha, B > 0$, there exist related machine instances $\mathcal{I}(\alpha, B)$ where* WSETF *is not $B$-competitive for weighted flow with a speed augmentation of $\alpha$.*

*Proof.* For this proof, let $\text{cost}(A)$ denote the total flow time of an algorithm $A$. The instance $\mathcal{I}$ is defined in the following manner and is similar to the lower bound on HDF. There is a "fast" machine of speed $S$, and infinitely many "slow" machines of speed 1. At time $t = 0$, a "heavy" job of weight $W$ and (unknown) length $L$ arrives. Then, at each time $\frac{L}{2\alpha S} + \frac{i}{\alpha S}$, for integer $0 \leq i < \frac{SL}{2}$, a "small" job of weight $w$ and length 1 arrives (all the parameters $S, W, L, w$ will be set appropriately when required). For notational convenience, we will assume that $\frac{SL}{2}$ is an integer.

At time $t = \frac{L}{2\alpha S}$, the priority of the heavy job is $\frac{W}{L/2} = \frac{2W}{L}$, since it has run on the fast machine and there is $\alpha$ speed augmentation for WSETF. Note that by definition, the priority of any job can only decrease over time. We now set the value of $w$ such that the worst-case priority of a small job (i.e., when it completes) is larger than this quantity. This implies that as long as a small job is unfinished, the heavy job can not run on the fast machine. The condition required for this is $\frac{w}{1} \geq \frac{2W}{L}$. We therefore set $w = \frac{2W}{L}$. Since each small job completes on the fast machine in $\frac{1}{\alpha S}$ time steps, a small job arrives as soon as its preceding small job is finished by WSETF. This will be repeated until all small jobs complete. This takes exactly $\frac{SL}{2}\frac{1}{\alpha S} = \frac{L}{2\alpha}$ units of time. This implies that the entire second half of the heavy job is processed on slow machines by WSETF. Thus, WSETF incurs a weighted flow time of at least $\text{cost}(\text{WSETF}) \geq W\frac{L}{2\alpha}$.

The optimal solution, however, will run the heavy job on the fast machine until completion, and run each small job on a *dedicated* unit speed machine. Recall that there are enough slow machines to accommodate all small jobs simultaneously. The cost of the optimal solution is then $\text{cost}(\mathsf{Opt}) = W\frac{L}{S} + \frac{SL}{2}w = W\frac{L}{S} + SW$. We set the length of the heavy job so that $W\frac{L}{S} = SW$, i.e., $L = S^2$. This implies a lower bound on the competitive ratio of $W\frac{L}{2\alpha}\Big/2W\frac{L}{S} = \frac{S}{4\alpha}$. To complete the proof, we simply set $S = 8\alpha B$.

## 2.3 A Lower Bound for Weighted Latest Arrival Processor Sharing (WLAPS)

Finally, in this section we show a lower bound on WLAPS. In order to simplify the presentation, we describe a lower bound instance for the Weighted Processor Sharing (WPS) algorithm (or Equipartition or Round-Robin) for total weighted flow time on related machines. Subsequently, we explain how this also translates to a lower bound for WLAPS. This is because WLAPS can be shown to always be dominated by WPS when WPS is given a constant amount of resource augmentation over WLAPS by definition of the algorithms. As usual, we begin with the algorithm description.

On a single fixed-speed machine, at any time, the algorithm WPS works on a job $J_j$ with weight $w_j$ at a speed of its "fair share", i.e., a fraction $\frac{w_j}{W}$ of the speed where $W$ is the total weight of unfinished jobs. How do we generalize this to multiple related machines? Ideally, we would like to process job $J_j$ at a rate of $\frac{w_j}{W}S$, where $S$ is the total speed of the fastest $n$ machines where $n$ is the number of unfinished jobs. However, this may not always be achievable. For example, if there is a single job $J_j$ with very large weight and $n-1$ jobs of negligible weight then job $J_j$ can only be processed at the speed of the fastest processor because a job can only be processed by a single processor at any point in time. This is much less than its fair share of the fastest $n$ processors. As a result, the most natural extension of WPS to the setting of heterogeneous processors, is to assume that each job $J_j$ is given $\frac{w_j}{W}$ share of the $\lfloor \frac{W}{w_j} \rfloor$ fastest processors. Therefore, job $J_j$ is processed with a total speed of $\displaystyle\sum_{i=1}^{\lfloor W/w_j \rfloor} \frac{w_j\, s_i}{W}$, where the machines are ordered in decreasing order of speed. It is not difficult to see that this scheduling policy can be achieved without scheduling a job on more than one machine at the same time. (Essentially, every job is scheduled to an extent of 1 across machines, and every machine has utilization of at most 1. Then we can decompose this fractional assignment into a convex combination of integer schedules, and then preemptively follow this combination). One can define the WLAPS algorithm in a similar fashion: WLAPS uses the WPS algorithm assuming that the latest $\epsilon$ fraction of the unsatisfied jobs that arrived the latest are the only jobs in the queue where $0 \leq \epsilon \leq 1$ is a constant that parametrizes WLAPS.

THEOREM 2.3. *For any constants $\alpha, B > 0$, there exist related machine instances $\mathcal{I}(\alpha, B)$ where* WPS *is not $B$-competitive for weighted flow with any constant speed augmentation $\alpha$.*

*Proof.* Consider the following instance. There are $n$ jobs, with job $J_j$ having a weight of $w_j = 1/j$ and length $l_j$ which will be determined later. There are $n$ machines with machine $i$ with speed $1/\sqrt{i}$. We will set the lengths of the jobs in such a way that all jobs complete at the same time in WPS.

We can bound the cost of WPS as follows. For such an instance, it is easy to see that the total speed at which job $J_j$ is processed by the WPS algorithm is exactly $\alpha \sum_{i=1}^{jH_n} \frac{1}{jH_n} \frac{1}{\sqrt{i}}$ where $\alpha$ is the speed augmentation WPS has over the optimal solution. Set $l_j$ to be precisely the above sum so that WPS completes all the jobs at exactly $t = 1$. Thus, WPS incurring a weighted flow time of $\sum_j w_j = H_n = \Omega(\log n)$.

Now we bound the cost of the optimal solution. Consider the following alternate schedule which simply schedules job $J_j$ on machine $j$. Noticing that $l_j = \alpha \frac{1}{jH_n} \sum_{i=1}^{jH_n} \frac{1}{\sqrt{i}} \leq O(\frac{\alpha}{\sqrt{jH_n}})$, we get that the weighted flow time for this schedule is at most

$$\sum_j \frac{w_j l_j}{s_j} = \sum_j \frac{1}{\sqrt{j}} l_j \leq \frac{O(\alpha)}{\sqrt{H_n}} \sum_j \frac{1}{j} \leq O(\alpha\sqrt{\log n})$$

which gives the $\Omega(\sqrt{\log n})$ bound on the competitive ratio of WPS for any speed augmentation of $\alpha$.

The careful reader might observe that the above algorithm does not utilize every machine to an extent of 1. Indeed, consider the example of one heavy job and a large number of very small jobs. Then the utilization of machines $3, 4, \ldots$ is negligibly small because each tiny job only occupies $w_j/W$ of these machines while much of the total weight in $W$ comes from the heavy job (even though the heavy job is never going to be scheduled on these slower machines). A better algorithm with possibly better performance is one where we re-weight the jobs' fair shares on each machine depending on which jobs have not yet been scheduled to full utilization. However, it can be shown that the above example has an unbounded competitive ratio even for this modified algorithm.

Now we show how the previous lemma extends to lower bound the performance of WLAPS.

COROLLARY 2.1. *For any constants $\alpha, B > 0$, there exist related machine instances $\mathcal{I}(\alpha, B)$ where* WLAPS *is not $B$-competitive for weighted flow with any constant speed augmentation $\alpha$.*

*Proof.* Note that by definition of WPS and WLAPS, when WPS is given a constant factor greater resource augmentation WPS schedule can only be better for total flow time than WLAPS. In particular, this holds when WPS is given more than a $\frac{1}{\epsilon}$ factor greater resource augmentation over WLAPS where $\epsilon$ is the constant that parametrizes WLAPS. Thus a lower bound of $c$ on the competitive ratio of WPS for any constant resource augmentation this implies a lower bound of $c$ on the competitive ratio of WLAPS for any constant resource augmentation.

## 3 LAPS on a Heterogeneous Multiprocessor for Flow Plus Energy

We now move on to our positive results. In particular, we show that a natural extension of the LAPS algorithm is scalable for the objective of minimizing the total flow time plus energy on a heterogeneous multiprocessor. Recall that in this model, each processor $i$ is speed-scalable with an independent speed function $Q_i$, and the scheduler at each time must decide on the speed-scaling policy and the job assignment policy. We begin by describing these policies for our algorithm LAPS.

**LAPS Speed Scaling Policy.** At each time $t$, a collection of processors and associated speed settings are selected to maximize the aggregate speed extracted, subject to the constraints that **(i)** the number of processors selected is at most $\lceil \epsilon |A(t)| \rceil$, and **(ii)** the aggregate power used is at most $\lceil \epsilon |A(t)| \rceil$ where $A(t)$ is the set of unfinished jobs for the online algorithm. More formally, the total speed extracted is given by the algorithm $\mathsf{GreedySS}(\epsilon |A(t)|)$ defined below. Note that if there are more machines than $\lceil \epsilon |A(t)| \rceil$ being used then our algorithm idles some of the processors even though there are jobs that could be scheduled.

**LAPS Job Selection Policy.** The extracted speed is evenly shared among the $\lceil \epsilon |A(t)| \rceil$ jobs that arrived the most recently. Such a distribution is possible because the number of machines running at non-zero speed in the speed scaling policy defined by GreedySS is at most $\lceil \epsilon |A(t)| \rceil$, and in this case, it is easy to have the algorithm cycle through different permutations to share the $\lceil \epsilon |A(t)| \rceil$ jobs on the chosen machines.

**The Speed Abstraction Problem and the** GreedySS **Algorithm.** We now define the speed extraction problem and define an optimal greedy algorithm GreedySS for this problem. The definition of the algorithm and proof of Lemma 3.1 appears in [13]. We re-state it for completeness.

**Speed Extraction Problem.** Given an integer power budget $W$, assign an integer power budget of $E_i$ to each processor $i$ so as to maximize the total extracted speed $\sum_i Q_i(E_i)$ subject to the constraints that $E_i$ is a non-negative integer, and $\sum_i E_i \leq W$.

**Algorithm** GreedySS**.** Intuitively the algorithm partitions the power budget into units, and assigns each unit to the machine which offers the best increase to the total speed that can be extracted. Note that we only constrain all feasible solutions for the above speed extraction problem to set integral values for the $E_i$'s, and make no such assumption about the different power settings of the machines in general. We now give the pseudo-code of GreedySS for completeness:

```
• Initially set $E_i := 0$ for all processors $i$. $E_i$ will eventually be the power used by processor $i$.
• For $j = 1$ to $W$ do
    • Let $k = \arg\max_i Q_i(E_i + 1) - Q_i(E_i)$
    • Increment $E_k$ to $E_k + 1$
• Set the speed $s_i$ of each processor $i$ to be $Q_i(E_i)$
```

LEMMA 3.1. *[13] The greedy algorithm* GreedySS *optimally solves the speed extraction problem.*

**3.1 Simplifying Assumptions** In order to convey the main idea of our analysis more clearly, we make the following simplifying assumptions. These assumptions will affect the resulting competitive ratio by a factor of at most $O_\epsilon(1)$.

(A): We assume that Opt is the GKP algorithm [14] which is a clairvoyant online algorithm that is $(1 + \epsilon)$-speed $O(1/\epsilon)$-competitive (by doing this, we only lose an additional factor of $O(1/\epsilon)$ in the competitive ratio). In particular, we crucially use the following property of the GKP algorithm: if GKP has $|O(t)|$ jobs unsatisfied at any time $t$, then the most speed it can use (in total over all machines) at this time is GreedySS$(|O(t)|)$. This follows from the fact that the GKP algorithm always runs any machine at a power that is at most the number of unfinished jobs assigned to the machine; this gives a valid solution for the Speed Extraction problem, and the quantity GreedySS$(|O(t)|)$ can only be larger by its optimality.

(B): We assume that the arrival times of jobs are distinct to simplify the analysis—we can handle identical arrivals by making infinitesimally small perturbations in the arrival times.

(C): We assume that LAPS is given $(1 + 10\epsilon)$ speed-up for some given parameter $0 < \epsilon < 1/10$.

**3.2 Potential Function Analysis** In this section we define and analyze a potential function to bound the competitiveness of LAPS. A tutorial on the use of potential functions to analyze scheduling problems can be found in [16]. Before we define the potential function, we introduce some notation. Denote the completion time of job $J_i$ as $C_i^A$ (and $C_i^O$) for the online algorithm (and optimal schedule respectively). At any time $t$, let $A(t)$ denote the set of unsatisfied jobs in the algorithm's schedule, and likewise $O(t)$ is the set of unsatisfied jobs in Opt's schedule. We also let $p_i^A(t)$ denote the remaining work at time $t$ for job $J_i$ in the algorithm's schedule, and $p_i^O(t)$ is the remaining work at time $t$ for job $J_i$ in Opt's schedule. Also define $z_i(t) = \max\{p_i^A(t) - p_i^O(t), 0\}$. For a job $J_i$, let rank$(i,t) := \sum_{j_{i'} \in A(t), r_{i'} \leq r_i} 1$ denote the number of unfinished jobs that arrived earlier. For any integer value $W$, let $Q(W) :=$ GreedySS$(W)$ denote the value of the optimal solution to the Speed Extraction problem with budget $W$. Our potential function is defined as follows.

$$\Phi(t) = \frac{2}{\epsilon^2} \sum_{J_i \in A(t)} \frac{\text{rank}(i,t) z_i(t)}{Q(\text{rank}(i,t))}$$

Now we bound the changes in the potential function. When bounding the changes, the following lemma will be useful. The proof of the following lemma is straightforward given the definition of $Q$.

LEMMA 3.2. *For any integers A and B such that $B \geq A$, we have that $Q(A) \geq \frac{A}{B} Q(B)$.*

*Proof.* Consider the run of the algorithm GreedySS$(B)$, and consider the $B$ increments that it made. By definition of GreedySS, $Q(B)$ is the sum of the incremental speeds we obtained at each step, and these values are monotonically non-increasing. As a result, if we only consider the first $A$ of these increments, we get a feasible solution to the Speed Extraction Problem on input $A$, and this has value at least $(A/B)Q(B)$.

The next two corollaries follow immediately from the above Lemma.

COROLLARY 3.1. *For any integer $i \geq 2$, we have that $\frac{i-1}{Q(i-1)} \leq \frac{i}{Q(i)}$*

COROLLARY 3.2. *For any integer $n$ and $0 < \epsilon \leq 1$, $Q(\lceil \epsilon n \rceil) \geq \epsilon Q(n)$.*

We are now ready to proceed with an amortized analysis. Let $\lambda > 0$ be some constant. Our aim is to show the following equation holds at all times $t$:

$$2|A(t)| + \frac{\mathrm{d}}{\mathrm{dt}}\Phi(t) \leq 2\lambda|O(t)|. \tag{3.1}$$

We will also show that $\Phi(0) = \Phi(\infty) = 0$, and $\Phi$ does not experience any increase at discontinuities. By integrating over time, we can then conclude that the total cost of the online algorithm (flow time plus energy) is at most $\lambda$ times that of the optimal algorithm. We now consider various cases:

**Job Arrival:** Consider when job $J_i$ arrives. This job has the largest rank out of all the jobs in $A(t)$. When $J_i$ arrives the rank of every other job remains the same and the terms in $\Phi$ corresponding to other jobs do not change. There is a new term added to the potential function corresponding to job $J_i$, but we know that $z_i(r_i) = 0$. Hence there is no overall change to the potential function value.

**Job Completion:** Consider a time $t$ when job $J_i$ completes in the online algorithm. The term in the potential function corresponding to $J_i$ must be 0 since $z_i(C_i) = 0$ by definition. This term drops out of the potential function, causing no change in the potential value. The ranks of all the other jobs which arrive after $J_i$ will decrease by 1, but by Corollary 3.1, the net change for these terms is negative. Therefore the completion of a job $J_i$ may cause a discontinuity at $\Phi(t)$, but we have ensured that $\Phi$ does not increase. Further, it can be seen that when Opt completes a job there is no effect on the potential function.

**Job Processing:** Here we consider the change in $\Phi$ due to the processing of jobs by the algorithm and the optimal solution in an infinitesimally small time interval $[t, t+\mathrm{d}t)$ when there are no job arrivals or completions. We will break the analysis into two cases.

`Case (a)`: $|O(t)| \geq \epsilon^2|A(t)|$. In this case, we ignore the change in $\Phi$ due to the algorithm's processing. This can be justified since the algorithm's processing can only decrease $\Phi$. We will charge the algorithm's flow time and any increase in the potential function directly to the optimal solution. We first upper bound the increase in $\Phi$. To this end, recall that the most speed Opt uses is $Q(|O(t)|)$ because assumption (A) states that Opt is the GKP algorithm. By Corollary 3.1, the adversary can increase $\Phi$ the most by working on the job with the highest rank. Let $|O(t)| = c|A(t)|$ where $c \geq \epsilon^2$. We obtain the following upper bound on the increase in $\Phi$ due to Opt's processing:

$$\frac{2}{\epsilon^2}\frac{|A(t)|}{Q(|A(t)|)}Q(|O(t)|) = \frac{2}{\epsilon^2}|A(t)|\frac{Q(c|A(t)|)}{Q(|A(t)|)}$$

There are two cases. If $c \geq 1$ then we appeal to Lemma 3.2 and infer that $\frac{2}{\epsilon^2}|A(t)|\frac{Q(c|A(t)|)}{Q(|A(t)|)} \leq \frac{2}{\epsilon^2}|A(t)|c = \frac{2}{\epsilon^2}|O(t)|$. Otherwise, $\frac{2}{\epsilon^2}|A(t)|\frac{Q(c|A(t)|)}{Q(|A(t)|)} \leq \frac{2}{\epsilon^2}|A(t)| \leq \frac{2}{\epsilon^4}|O(t)|$ since $Q$ is non-decreasing and $|A(t)| \leq \frac{1}{\epsilon^2}|O(t)|$. Thus the increase is at most a constant times the optimal solution's current cost. This bound combined with the fact that the algorithm's cost is at most $2|A(t)| \leq \frac{2}{\epsilon^2}|O(t)|$, we get that the term $2|A(t)| + \frac{\mathrm{d}}{\mathrm{dt}}\Phi(t)$ is at most $\frac{4}{\epsilon^4}|O(t)|$. Thus, setting the constant $\lambda$ from above to be $2/\epsilon^4$ suffices.

`Case (b)`: $|O(t)| \leq \epsilon^2|A(t)|$. In this case, we need to use the potential function to pay for the increase in the algorithm's objective. First consider the change in $\Phi$ due to the adversary's processing of jobs. Again by assumption (A), the most speed Opt can use at time $t$ is $Q(|O(t)|)$. By Corollary 3.1, the largest increase in the potential function would occur when Opt uses all of the power invested on the job with the highest rank. Therefore the largest increase in the potential due to Opt's processing is upper bounded by:

$$\begin{aligned} & \frac{2}{\epsilon^2}\frac{|A(t)|Q(|O(t)|)}{Q(|A(t)|)} \\ \leq\ & \frac{2}{\epsilon^2}\frac{|A(t)|Q(\lceil\epsilon|A(t)|\rceil)}{Q(|A(t)|)} \\ & \text{[By definition of } Q \text{ and } |O(t)| \leq \epsilon^2|A(t)| \leq \lceil\epsilon|A(t)|\rceil] \end{aligned}$$

Now consider the change in the potential function due to the algorithm's processing. Again, by the definition of our algorithm, we know that the algorithm round robins the $\lceil\epsilon|A(t)|\rceil$ latest arriving jobs on at most $\lceil\epsilon|A(t)|\rceil$ machines whose total speed extracted is $Q(\lceil\epsilon|A(t)|\rceil)$. Let $A'(t)$ be the set of jobs that the algorithm processes. For any job $J_i$ which the algorithm processes, $\text{rank}(i,t) \geq (1-\epsilon)|A(t)|$ and $Q(\text{rank}(i,t)) \leq Q(|A(t)|)$. Further, we know that the $z$ variables decrease for at least $\lceil\epsilon|A(t)|\rceil - \epsilon^2|A(t)|$ jobs since the optimal solution has at most $\epsilon^2|A(t)|$ jobs in its queue by assumption. For these jobs $z_i$ decreases at a rate of $\frac{1}{\epsilon|A(t)|}Q(\lceil\epsilon|A(t)|\rceil)(1+10\epsilon)$ using the fact that the algorithm is given $(1+10\epsilon)$ resource augmentation and the definition of the algorithm. Thus we have that the change in $\Phi$ due to the algorithm's processing is at most the following

$$\begin{aligned} & -\frac{2}{\epsilon^2}\sum_{J_i \in A'(t)\setminus O(t)}\frac{\text{rank}(i,t)}{Q(\text{rank}(i,t))} \\ & \cdot\frac{1}{\lceil\epsilon|A(t)|\rceil}Q(\lceil\epsilon|A(t)|\rceil)(1+10\epsilon) \\ \leq\ & -\frac{2}{\epsilon^2}\sum_{J_i \in A'(t)\setminus O(t)}\frac{(1-\epsilon)|A(t)|}{Q(\text{rank}(i,t))} \\ & \cdot\frac{1}{\lceil\epsilon|A(t)|\rceil}Q(\lceil\epsilon|A(t)|\rceil)(1+10\epsilon) \\ & \text{[Since rank}(i,t) \geq (1-\epsilon)|A(t)| \text{ for } J_i \in A'(t)] \end{aligned}$$

$$\begin{aligned}
&\le -\frac{2}{\epsilon^2}\cdot\frac{(1-\epsilon)|A(t)|}{Q(|A(t)|)} \\
&\quad \cdot\frac{1}{\lceil\epsilon|A(t)|\rceil}\sum_{J_i\in A'(t)\setminus O(t)} Q(\lceil\epsilon|A(t)|\rceil)(1+10\epsilon) \\
&\quad \text{[Since } Q(|A(t)|)\ge Q(\text{rank}(i,t)) \text{ for all } J_i\in A(t)\text{]} \\
&\le -\frac{2}{\epsilon^2}\cdot\frac{(1-\epsilon)|A(t)|}{Q(|A(t)|)}\cdot\frac{1}{\lceil\epsilon|A(t)|\rceil} \\
&\quad \cdot\left(\lceil\epsilon|A(t)|\rceil-\epsilon^2|A(t)|\right)Q(\lceil\epsilon|A(t)|\rceil)(1+10\epsilon) \\
&\quad \text{[Since } |A'(t)|=\lceil\epsilon|A(t)|\rceil\text{]} \\
&\le -\frac{2}{\epsilon^2}\cdot\frac{(1-\epsilon)^2|A(t)|}{Q(|A(t)|)}\cdot Q(\lceil\epsilon|A(t)|\rceil)(1+10\epsilon)
\end{aligned}$$

Thus the total net change in the potential function is at most,

$$\begin{aligned}
&\frac{2}{\epsilon^2}\frac{|A(t)|Q(\lceil\epsilon|A(t)|\rceil)}{Q(|A(t)|)} \\
&-\frac{2}{\epsilon^2}\frac{(1-\epsilon)^2|A(t)|Q(\lceil\epsilon|A(t)|\rceil)(1+10\epsilon)}{Q(|A(t)|)} \\
\le\ & -\frac{2}{\epsilon}|A(t)|\frac{Q(\lceil\epsilon|A(t)|\rceil)}{Q(|A(t)|)} \quad \text{[Since } \epsilon\le 1/10\text{]} \\
\le\ & -2|A(t)| \quad \text{[By Corollary 3.2]}
\end{aligned}$$

Thus, the net change in the potential function plus the increase in the algorithm's objective is non-positive. So this gives us the restriction that $\lambda \ge 0$. Therefore, we get that $\lambda = \frac{2}{\epsilon^4}$ suffices in all cases.

For the final analysis, we add the upper bound on the change for each of the cases we studied above. Let $\frac{\mathrm{d}}{\mathrm{dt}}\Phi(t)$ denote the change (rate) of $\Phi(t)$, $\frac{\mathrm{d}}{\mathrm{dt}}\text{LAPS}(t)$ denote the change of our algorithm's objective and $\frac{\mathrm{d}}{\mathrm{dt}}\mathsf{Opt}(t)$ denote the change in the optimal solution's objective. We have that $\frac{\mathrm{d}}{\mathrm{dt}}\text{LAPS}(t)+\frac{\mathrm{d}}{\mathrm{dt}}\Phi(t)\le\frac{2}{\epsilon^4}\frac{\mathrm{d}}{\mathrm{dt}}\mathsf{Opt}(t)$ by the previous arguments. Thus,

$$\begin{aligned}
\text{LAPS} &= \int_0^\infty \Big(\frac{\mathrm{d}}{\mathrm{dt}}\text{LAPS}(t)\Big)\mathrm{dt} \\
&= \int_0^\infty \Big(\frac{\mathrm{d}}{\mathrm{dt}}\text{LAPS}(t)+\frac{\mathrm{d}}{\mathrm{dt}}\Phi(t)\Big)\mathrm{dt} \\
&\qquad \text{[Since } \Phi(0)=\Phi(\infty)=0\text{]} \\
&\le \int_0^\infty \Big(\frac{2}{\epsilon^4}\frac{\mathrm{d}}{\mathrm{dt}}\mathsf{Opt}(t)\Big)\mathrm{dt} = \frac{2}{\epsilon^4}\mathsf{Opt}
\end{aligned}$$

However, since we assumed that Opt runs the GKP algorithm (which is in itself $(1+\epsilon)$-speed $O(1/\epsilon)$-competitive from [14]), we get that the overall competitive ratio of our non-clairvoyant algorithm is $O(1/\epsilon^5)$. We stress that we have not tried to optimize the competitive ratio but rather to show that the related machines setting admits a non-clairvoyant scalable algorithm.

THEOREM 3.1. *The algorithm LAPS is $(1+\epsilon)$-speed $O(1/\epsilon^5)$-competitive for the problem of flow time plus energy on related machines.*

## References

[1] Lachlan L. H. Andrew, Minghong Lin, and Adam Wierman. Optimality, fairness, and robustness in speed scaling designs. In *SIGMETRICS*, pages 37–48, 2010.

[2] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *SODA*, pages 693–701, 2009.

[3] Nikhil Bansal, Ravishankar Krishnaswamy, and Viswanath Nagarajan. Better scalable algorithms for broadcast scheduling. In *ICALP (1)*, pages 324–335, 2010.

[4] Nikhil Bansal and Kirk Pruhs. Server scheduling to balance priorities, fairness, and average quality of service. *SIAM J. Comput.*, 39(7):3311–3335, 2010.

[5] Nikhil Bansal, Kirk Pruhs, and Clifford Stein. Speed scaling for weighted flow time. *SIAM J. Comput.*, 39(4):1294–1308, 2009.

[6] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Online weighted flow time and deadline scheduling. *Journal of Discrete Algorithms*, 4(3):339–352, 2006.

[7] F.A. Bower, D.J. Sorin, and L.P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *Micro, IEEE*, 28(3):17 –25, may-june 2008.

[8] Jivitej S. Chadha, Naveen Garg, Amit Kumar, and V. N. Muralidhara. A competitive algorithm for minimizing weighted flow time on unrelatedmachines with speed augmentation. In *Symposium on Theory of Computing*, pages 679–684, 2009.

[9] Ho-Leung Chan, Jeff Edmonds, Tak Wah Lam, Lap-Kei Lee, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Nonclairvoyant speed scaling for flow and energy. In *STACS*, pages 255–264, 2009.

[10] Sze-Hang Chan, Tak Wah Lam, and Lap-Kei Lee. Non-clairvoyant speed scaling for weighted flow time. In *ESA (1)*, pages 23–35, 2010.

[11] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 685–692, 2009.

[12] Kyle Fox and Benjamin Moseley. Online scheduling on identical machines using SRPT. In *SODA*, pages 120–128, 2011.

[13] Anupam Gupta, Ravishankar Krishnaswamy, and Kirk Pruhs. Nonclairvoyantly scheduling power-heterogeneous processors. In *Green Computing Conference*, 2010.

[14] Anupam Gupta, Ravishankar Krishnaswamy, and Kirk Pruhs. Scalably scheduling power-heterogeneous processors. In *ICALP (1)*, pages 312–323, 2010.

[15] Nikos Hardavellas. Exploiting dark silicon for energy efficiency. Article to appear in IEEE Micro.

[16] Sungjin Im, Benjamin Moseley, and Kirk Pruhs. A tutorial on amortized local competitiveness in online scheduling. *SIGACT News*, 42:83–97, June 2011.

[17] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.

[18] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 64 – 75, june 2004.

[19] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM.

[20] R. Merritt. CPU designers debate multi-core future. EE Times, Feb. 2010.

[21] Tomer Y. Morad, Uri C. Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Comput. Archit. Lett.*, 5:4–, January 2006.

[22] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. *Theor. Comput. Sci.*, 130(1):17–47, 1994.

[23] Cynthia A. Phillips, Clifford Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.

[24] Kirk Pruhs, Jiri Sgall, and Eric Torng. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter Online Scheduling. 2004.

[25] Eric Torng and Jason McCullough. SRPT optimally utilizes faster machines to minimize flow time. *ACM Trans. Algorithms*, 5:1:1–1:25, December 2008.

## A Summary of Known Results and Our Results

We summarize previously known results together with our results. Our results are marked by [*]. See the following summary to see what scheduling algorithm each short name refers to. Any algorithm that starts with "W" implies the weighted version of its unweighted counterpart. Note that showing an upper bound for the non-clairvoyant setting subsumes the clairvoyant setting. Likewise, an upper bound for the speed scaling setting subsumes the fixed processor setting. A lower bound for an algorithm in the fixed processor setting then implies a lower bound for the algorithm in the speed scaling setting.

- PS: Round Robin shares the total processing speed amount the jobs proportionally to the weight of the jobs.

- SETF: Shortest Elapsed Time First gives the $i^{th}$ fastest processor to the job with $i^{th}$ highest apparent density, where the apparent density is the weight of the job over the amount the job has been processed. Ties are broken by sharing the processor.

- LAPS: Latest Arrival Processor Sharing applies Round Robin to the latest arriving $\epsilon$ fraction of the jobs.

- SRPT: Shortest Remaining Processing Time gives the $i^{th}$ fastest processor to the job with the $i^{th}$ least remaining work.

- HDF: Highest Density First gives the $i^{th}$ fastest processor to the job with the $i^{th}$ highest density, where density is weight over original work.

- SJF: Shortest Job First gives the $i^{th}$ fastest processor to the job with the $i^{th}$ least original work.

| | Fixed Speed Processor | | Speed-Scalable Processor | |
|---|---|---|---|---|
| | Clairvoyant | Non-Clairvoyant | Clairvoyant | Non-Clairvoyant |
| Unweighted | SRPT optimal<br>SJF scalable [6] | SETF scalable [17]<br>LAPS scalable [11] | SRPT competitive [5, 2, 1]<br>SJF scalable [2] | LAPS scalable [9] |
| Weighted | HDF scalable [6] | WSETF scalable [4]<br>WLAPS scalable [3] | HDF scalable [2] | WLAPS scalable [10] |

Figure 1: Guarantees for the standard scheduling algorithms on a uniprocessor

| | Fixed Speed Processors | |
|---|---|---|
| | Clairvoyant | Non-Clairvoyant |
| Unweighted | SRPT scalable [25, 12]<br>SJF scalable [25] | LAPS scalable [11] |
| Weighted | HDF scalable [25] | WLAPS scalable [3] |

Figure 2: Guarantees for the standard scheduling algorithms on a homogeneous multiprocessor

| | Fixed Speed Processors | | Speed-Scalable Processors | |
|---|---|---|---|---|
| | Clairvoyant | Non-Clairvoyant | Clairvoyant | Non-Clairvoyant |
| Unweighted | | | | PS $(2+\epsilon)$-speed<br>$O(1)$-competitive [13]<br>LAPS Variant<br>scalable [*] |
| Weighted | HDF not scalable [*]<br>Scalable Algorithm [8] | WSETF not scalable [*]<br>WLAPS not scalable [*] | Scalable Algorithm [14] | |

Figure 3: Guarantees for the standard scheduling algorithms on a heterogeneous multiprocessor

## B Local Competitiveness Lower Bounds

A scheduling algorithm $A$ is said to be locally competitive if the number (or total weight) of unfinished jobs at any time $t$ under $A$'s schedule is comparable to the number (or total weight) of unfinished jobs in the optimal schedule. Local competitiveness implies that the algorithm's competitive ratio can be bounded for (weighted) flow time because the (weight) number of the unsatisfied jobs in a schedule at some time is the instantaneous increase in the objective at that time. Many scheduling algorithms have been proved to have bounded competitiveness using a local competitiveness argument. In particular, SRPT and SJF can be shown to be scalable on identical parallel machines via a local competitiveness argument. We however show that these algorithms are not locally competitive on related machines even with any constant speedup. Recall that in the related machines setting machines/processors have different fixed speeds and consume not power. We show that local competitiveness cannot be shown even in the unweighted setting.

THEOREM B.1. *For any $s \geq 1$, assume that* SRPT *or* SJF *is given $s$-speed augmentation. Then there exists a schedule and time $t$ such that the schedule finished all jobs at time $t$ while* SRPT *or* SJF *has unsatisfied jobs.*

*Proof.* We first describe the instance. There are $k + 1$ groups of machines, $\mathcal{M}_0, \mathcal{M}_1, ..., \mathcal{M}_k$. Group $\mathcal{M}_i$ has $h^{2(k-i)}$ machines of speed $h^i$ where $h$ is a sufficiently large constant. There are $k + 1$ groups of jobs, $\mathcal{J}_0, \mathcal{J}_1, ..., \mathcal{J}_k$. Group $\mathcal{J}_i$ has $h^{2(k-i)}$ jobs of size $h^i$. For notational convenience, we will use subscript to denote a subset of groups. For example, $\mathcal{M}_{\geq i} = \bigcup_{i' \geq i} \mathcal{M}_{i'}$.

Note that one can finish all jobs by time 1 by scheduling each job in $\mathcal{J}_i$ on one machine in $\mathcal{M}_i$. We will show that SRPT cannot finish all jobs by time $\frac{k}{s}$. Then the theorem follows by setting $k = s + 1$ and $t = 1$. At time 0, SRPT fills all machines in $\mathcal{M}_{\geq 1}$ with the jobs in $\mathcal{J}_0$. This is because jobs in $\mathcal{J}_0$ are the shortest jobs and $|\mathcal{J}_0| = h^{2k} > \sum_{i=1}^{k} h^{2(k-i)} = |\mathcal{M}_{\geq 1}|$. Until time $\frac{1}{2s}$, no job in $\mathcal{J}_0$ can be finished by SRPT unless it is processed on one of the machines in $\mathcal{M}_{\geq 1}$. The total volume of jobs in $\mathcal{J}_0$ that can be processed on $\mathcal{M}_{\geq 1}$ for $\frac{1}{2s}$ time units by $s$ speed resource augmented machines is at most $\frac{1}{2}\sum_{i=1}^{k} h^{2(k-i)} h^i = \frac{1}{2}\sum_{i=1}^{k} h^{2k-i} \leq h^{2k-1}$. The last inequality holds for sufficiently large $h$. Further, machines in $\mathcal{M}_0$ can process at most $h^{2k}/2$ volume of jobs in $\mathcal{J}_0$ during $[0, \frac{1}{2s}]$. Hence we have a lower bound $h^{2k} - h^{2k}/2 - h^{2k-1} = h^{2k}/2 - h^{2k-1}$ on the total remaining volume of the unfinished jobs in $\mathcal{J}_0$ at time $\frac{1}{2s}$. This is because the total volume of jobs in $\mathcal{J}_0$ is $h^{2k}$ and a total volume of at most $h^{2k}/2 + h^{2k-1}$ can be processed during $[0, \frac{1}{2s}]$ by SRPT with $s$ resource augmentation. Since this lower bound is larger than $|\mathcal{M}_{\geq 1}|$, we know that during $[0, \frac{1}{2s}]$, all jobs in $\mathcal{J}_{\geq 1}$ were scheduled only on machines in $\mathcal{M}_0$. It is easy to see that each job in $\mathcal{J}_{\geq 1}$ has been processed by a fraction of at most $\frac{1}{h}$.

The remaining proof can be completed by repeating this argument. Formally, one can show the following: At time $\frac{\ell}{2s}$ for integer $1 \leq \ell \leq k$, each job in $\mathcal{J}_{\geq \ell}$ has remaining size that is at least $(1 - 1/h)^{\ell}$ times its initial size. The proof for SJF is the same, since SJF and SRPT produce the same schedule on any instance where all jobs arrive at the same time by definition of the algorithms.