Multiprocessor jobs, preemptive schedules, and one-competitive online algorithms

Jiří Sgall[1] and Gerhard J. Woeginger[2]

[1] Computer Science Institute of Charles University, Praha, Czech Republic, sgall@iuuk.mff.cuni.cz.
[2] Department of Mathematics and Computer Science, TU Eindhoven, Netherlands, gwoegi@win.tue.nl

Abstract. We study online preemptive makespan minimization on m parallel machines, where the (multiprocessor) jobs arrive over time and have widths from some fixed set $W \subseteq \{1, 2, \ldots, m\}$. For every number m of machines we concisely characterize all the sets W for which there is a 1-competitive *fully* online algorithm and all the sets W for which there is a 1-competitive *nearly* online algorithm.

1 Introduction

In a multiprocessor job system, jobs may occupy several machines in parallel. For instance, concurrent threads of some parallel application may be run simultaneously and thereby block several machines. Multiprocessor scheduling problems have been studied extensively over the last three decades. The papers [4, 5] by Drozdowski provide an excellent introduction to the area.

The considered scheduling model. We investigate the following problem of scheduling n multiprocessor jobs $J_1, \ldots, J_n$ on m identical machines. Every job J_j $(j = 1, \ldots, n)$ has a *length* $p(J_j)$, a *width* $w(J_j)$, and a *release date* $r(J_j)$. Job J_j enters the system at time $r(J_j)$, and requests simultaneous processing on exactly $w(J_j)$ machines for a total of $p(J_j)$ time units. Preemption is allowed: the processing of a job can be interrupted at any moment in time, and can be resumed at any later moment on the same set of machines or on another set of machines, however this set always has to contain exactly $w(J_j)$ machines. Every machine can process at most one job at a time. The goal is to minimize the largest job completion time, which is called the *makespan* of the schedule. In the three-field notation this problem is denoted $P|\text{pmtn}, \text{size}_j, r_j|C_{\max}$.

The computational complexity of this problem is well-understood. If the number m of machines is a fixed constant, then $Pm|\text{pmtn}, \text{size}_j, r_j|C_{\max}$ can be formulated as a linear program of polynomial size and hence is solvable in polynomial time; see Blazewicz, Drabowski & Weglarz [2]. If the number m of machines is part of the input, then the problem is NP-hard even in the absence of job release dates; see Drozdowski [3]. The survey paper [4] by Drozdowski summarizes the complexity landscape around scheduling multiprocessor jobs. On the

approximation side, Johannes [8] and independently Naroska & Schwiegelshohn [11] show that the List Scheduling algorithm yields an approximation ratio of 2.

Online algorithms. In the current paper we are mainly interested in the online version of $P|\text{pmtn}, \text{size}_j, r_j|C_{\max}$ where jobs arrive over time; see for instance Sgall [14] or Pruhs, Sgall & Torng [12] for surveys of the standard online scheduling scenarios. The scheduler learns about job J_j at time $r(J_j)$, and immediately receives full knowledge of the length and the width of the job. At any moment in time the online scheduler decides which of the available jobs should be processed on which of the machines. An online algorithm is *c-competitive*, if for all instances the online makespan is at most a multiplicative factor c above the optimal offline makespan. The *competitive ratio* of an online algorithm is the smallest real c for which the algorithm is c-competitive.

As jobs may be preempted, there arises a delicate distinction between *fully* online and *nearly* online algorithms. A *fully* online algorithm makes all its decisions based on the sole knowledge of the jobs that have arrived up to the current moment. A *nearly* online algorithm additionally knows the arrival time of the next job arriving in the future. As a nearly online algorithm has more information than a fully online algorithm, nearly online algorithms may possibly reach better competitive ratios than fully online algorithms. However, the actual difference between these two concepts is very small. If one allows some form of time-sharing, for example an infinite number of preemptions and infinitesimally small preempted job pieces, then fully online and nearly online algorithms are essentially equivalent: The best possible fully online competitive ratio then equals the best possible nearly online competitive ratio. If on the other hand one only allows a finite number of preemptions, then fully online algorithms in general are slightly weaker than nearly online algorithms. Nevertheless, whenever there exists a c-competitive nearly online algorithm, then for every $\varepsilon > 0$ there also exists a $(c + \varepsilon)$-competitive fully online algorithm.

Hong & Leung [7] construct a 1-competitive *fully* online algorithm for $P|\text{pmtn}, r_j|C_{\max}$, that is, for the variant where all widths are 1 and where every job requests processing on a single machine. (The online algorithm in [7] also knows the optimal offline makespan from the very beginning, but it only uses this knowledge to stop in an early stage if it detects an infeasibility.) Labetoulle, Lawler, Lenstra & Rinnooy Kan [9] give a 1-competitive *nearly* online algorithm for problem $Q|\text{pmtn}, r_j|C_{\max}$, where the machines are uniformly related. Vestjens [15] strengthens this result: he provides a concise analysis of $Q|\text{pmtn}, r_j|C_{\max}$ and characterizes all combinations of the machine speeds for which there exists a 1-competitive *fully* online algorithm. Very recently, Guo & Kang [6] constructed a 1-competitive *fully* online algorithm for the two-machine problem $P2|\text{pmtn}, \text{size}_j, r_j|C_{\max}$. On the negative side, Johannes [8] proves that no (fully or nearly) online algorithm for the general problem $P|\text{pmtn}, \text{size}_j, r_j|C_{\max}$ can have a competitive ratio below $6/5$.

Contribution of this paper. We analyze the problem of $P|\text{pmtn}, \text{size}_j, r_j|C_{\max}$ where all job widths belong to some fixed subset $W \subseteq \{1, 2, \ldots, m\}$ a priori known to the scheduler. For every number m of machines we characterize all the

sets W for which there is a 1-competitive *fully* online algorithm and all the sets W for which there is a 1-competitive *nearly* online algorithm.

This generalizes the two 1-competitive fully online algorithms mentioned above: the algorithm of Hong & Leung [7] which covers the cases with $W = \{1\}$ and the algorithm of Guo & Kang [6] which covers the case $W = \{1, 2\}$ and $m = 2$.

Statement of main result. For a number m of machines and a width w, we define the *rank* of w relative to m as $R(w, m) = \lfloor m/w \rfloor$. In other words, $R(w, m)$ denotes the maximum number of jobs of width w that can be processed simultaneously on m machines.

A width w is called *fat* for m machines if $w > m/2$ (so that w has rank 1), and it is called *skinny* if $w \leq m/2$ (so that w has rank at least 2). For a set $W \subseteq \{1, 2, \ldots, m\}$, we denote by $W^- \subseteq W$ its skinny elements and by $W^+ \subseteq W$ its fat elements. Jobs are called fat respectively skinny if their width is fat respectively skinny.

Theorem 1.1. *Let $m \geq 1$ be the number of machines and let $W \subseteq \{1, 2, \ldots, m\}$ be the set of possible job widths. There exists a 1-competitive* nearly *online scheduling algorithm on m identical parallel machines with job widths in W, if and only if the following two conditions are both fulfilled:*

(c1) *All $a, b \in W^-$ satisfy $R(a, m) = R(b, m)$; in other words, all skinny widths in W have the same rank relative to m.*

(c2) *All $a, b \in W^-$ and all $c \in W^+$ satisfy $R(a, m - c) = R(b, m - c)$; in other words, whenever a fat job blocks some of the machines, then all the skinny widths in W have the same rank relative to the number of remaining machines.*

Furthermore there exists a 1-competitive fully *online scheduling algorithm on m identical parallel machines with job widths in W, if and only if conditions (c1) and (c2) together with the following condition (c3) are fulfilled:*

(c3) *All $c \in W^+$ and all $a \in W^-$ satisfy $R(a, m - c) = 0$ or $R(a, m) = 2$.*

Note that conditions (c1) and (c2) guarantee that all the ranks are independent of a. Then condition (c3) gives only two possibilities: Either every fat job blocks execution of any other job, or no three jobs can be executed together (and then some fat jobs may block all other jobs, while other fat jobs may be scheduled together with any single skinny width job).

The rest of the paper is dedicated to the proof of Theorem 1.1. After providing some technical preliminaries, the four Sections 2–5 contain the proofs of the if-parts and the only-if-parts for the characterization of the nearly and the fully online case.

Technical preliminaries. This section collects some tools and observations that will be useful in the rest of the paper. Preemptive makespan minimization of jobs with unit-widths on parallel machines (that is, problem $P|\text{pmtn}|C_{\max}$) can be solved by the wrap-around rule of McNaughton [10]. We will apply McNaughton's classical result in the following equivalent formulation for multiprocessor jobs.

Proposition 1.2. *Consider a system with m machines and n multiprocessor jobs $J_1, \ldots, J_n$, where all jobs are available at time 0 and where all job widths have the same rank R relative to m. There is a preemptive schedule that completes all jobs by time t, if and only if*

(i) *The length of every job satisfies $p(J_j) \leq t$.*
(ii) *The total length of all jobs satisfies $\sum_{j=1}^{n} p(J_j) \leq Rt$.*

Hong & Leung [7] gave a 1-competitive fully online algorithm for the special case of $P|\text{pmtn}, \text{size}_j, r_j|C_{\max}$ where all jobs have width 1. We will use the following equivalent formulation of their result for multiprocessor jobs.

Proposition 1.3. *The online problem of preemptively scheduling multiprocessor jobs on m machines allows a fully online 1-competitive algorithm, if all job widths have the same rank R relative to m.*

Finally, we state some observations on job widths and job ranks. If two widths a and b have the same rank r relative to m, then any combination of r jobs of width a or b can be run in parallel on m machines. We will use the following observation many times implicitly in our arguments.

Observation 1.4. *For $a, b \in \{1, 2, \ldots, m\}$ with $R(a, m) = R(b, m)$ it holds that*

(i) *$b < 2a$ and $a < 2b$;*
(ii) *$R(b, m - ka) = R(b, m) - k$ for $k = 1, \ldots, R(a, m)$.*

2 The negative result for nearly online algorithms

In this section we prove that whenever a set W of job widths violates one of the conditions (c1) and (c2) in the statement of Theorem 1.1, then no 1-competitive *nearly* online scheduling algorithm can exist on m machines.

All our arguments are centered around the *utilization* of machines in an adversarially constructed instance: At any moment in time, the total width of the jobs run in an optimal schedule will be at least the total width of the jobs run in the online schedule, and on some non-trivial time interval the optimal total width will be strictly larger. Consequently the online makespan will be strictly worse than the optimal makespan, and the nearly online scheduler cannot be 1-competitive.

Due to the space limit we present only the case when the condition (c1) is violated. The proof when (c2) is violated, i.e., when $R(a, m - c) > R(b, m - c)$ works similarly, as we can simply block c machines by a fat job. An exception is the case when $R(b, m - c) \in \{0, 1\}$, which needs a separate construction.

Throughout this section we assume that condition (c1) is violated, and we consider $a, b \in W^-$ with $R(a, m) > R(b, m) > 1$; note that this implies $b > a$. For simplicity of presentation we introduce $r = R(b, m)$. The proof of the next lemma is omitted.

Lemma 2.1. *Let x_0 and y_0 be integers that maximize the value of $ax+by$ subject to the constraints $ax+by \leq m$ and $x, y \geq 0$. Furthermore let x_1 and y_1 be integers that maximize the value of $ax + by$ subject to the constraints $ax + by \leq m$ and $x \geq 1$ and $y \geq 0$. Then $x_0 \geq 1$ or $x_1 \geq 2$.*

We present an adversarial argument against an arbitrary nearly online scheduler, which is built around x_0, y_0, x_1, y_1 from Lemma 2.1. The first adversarial phase is as follows: at time 0 we confront the scheduler with $R(a, m - rb) + 1$ jobs of width a and length r, and with $(r-1)r$ jobs of width b and length 1. The second adversarial phase starts at time $r - 1$, when a group of long jobs arrives that all have the same length $L = r + R(a, m - rb)$. If $x_0 \geq 1$ (see Lemma 2.1), then this group consists of $x_0 - 1$ jobs of width a and of y_0 jobs of width b. If $x_0 = 0$, then this group consists of $x_1 - 1 \geq 1$ jobs of width a and of y_1 jobs of width b.

The optimal schedule S_1 for the first phase continuously processes all the $R(a, m - rb) + 1$ jobs of width a together with some $r - 1$ of the jobs of width b. The makespan of this schedule S_1 equals r. The optimal schedule S_2 for all the jobs from both phases is as follows:

- During the time interval $[0, r - 1]$, schedule S_2 continuously processes $R(a, m - rb)$ jobs of width a (from the first phase) together with some r of the jobs of width b (from the first phase). Then at time $r - 1$ all jobs from the first phase are completed, with the sole exception of $r + R(a, m - rb)$ unprocessed time units of the jobs of width a.
- From time $r - 1$ to time $r - 1 + L$, schedule S_2 continuously processes x_0 jobs of width a and y_0 jobs of width b (in the case where $x_0 \geq 1$) or it continuously processes x_1 jobs of width a and y_1 jobs of width b (in the case where $x_0 = 0$). In either case the makespan of S_2 equals $r - 1 + L$.

During $[0, r - 1]$, schedule S_2 utilizes $a \cdot R(a, m - rb) + br$ machines. During the remaining time, schedule S_2 either utilizes $ax_0 + by_0$ machines (in the case $x_0 \geq 1$) or $ax_1 + by_1$ machines (in the case $x_0 = 0$). According to Lemma 2.1, this is either the globally best possible utilization (if $x_0 \geq 1$), or the best possible utilization subject to the constraint that at least one job of width a is running (if $x_0 = 0$).

How would a 1-competitive nearly online algorithm behave on this instance? As it has no knowledge on the jobs from the second phase, the online algorithm would have to follow the structure of the optimal schedule S_1 during the time interval $[0, r - 1]$. Then it utilizes $a \cdot (R(a, m - rb) + 1) + b \cdot (r - 1)$ machines, which is strictly smaller than the utilization of schedule S_2 during $[0, r - 1]$. If $x_0 \geq 1$, then during the remaining time the online scheduler cannot beat the (globally optimal) utilization of schedule S_2. If $x_0 = 0$, then the online scheduler must process one of the $x_1 - 1 \geq 1$ jobs of width a (from the second phase) from time $r - 1$ to time $r - 1 + L$; by Lemma 2.1 there is no way of beating the utilization of schedule S_2 in this case. All in all, the utilization of the online schedule is sometimes weaker but never better than that of the optimal schedule.

3 The positive result for nearly online algorithms

In this section we design and analyze the 1-competitive *nearly* online scheduling algorithm FatMcN for all cases where the job widths in W satisfy conditions (c1) and (c2) in Theorem 1.1. Our approach applies and extends the machinery

from the area, as introduced for instance in Schmidt [13], Hong & Leung [7], and Albers & Schmidt [1]. From the technical point of view our arguments are more subtle, and our results seem to reach the very limits of what can be derived for this type of online problem.

Description of the algorithm. The main idea of our nearly online algorithm FatMcN is (i) to handle the fat jobs (Fat) with highest preference, and (ii) to fit the skinny jobs into the remaining space by using McNaughton's result (Mc) in Proposition 1.2.

Whenever there are fat jobs available, FatMcN selects a fat job of maximum width and runs it. The resulting completion times of the fat jobs together with the arrival times of all (fat or skinny) jobs constitute the so-called *critical* time points $0 = t_0 < t_1 < \cdots < t_s = C_{\max}$. (For technical reasons, we will assume that at the very end of the instance a final trivial job of length 0 is released, so that the last critical time point coincides with the optimal makespan.) For every time slot $[t_k, t_{k+1}]$ we define m_k as the maximum number of skinny jobs that can be processed simultaneously during the slot. Note that by conditions (c1) and (c2) the numbers m_k are well-defined, and that in fact *any* collection of m_k skinny jobs can be processed simultaneously at any time point during the slot. As a nearly online algorithm, FatMcN is always aware of the next critical time point.

The schedule for the skinny jobs during time slot $[t_k, t_{k+1}]$ is determined at time t_k. Let $p_1 \geq p_2 \geq \cdots \geq p_s$ denote the processing times of the skinny job pieces that are available and still need processing at time t_k. Intuitively it is clear that long job pieces should receive more processing than short job pieces. To make this intuition precise, we introduce a threshold τ whose exact value will be fixed later.

- Short job pieces with $p_j \leq \tau$ are not processed during the time slot.
- Long job pieces with $p_j > \tau$ are processed for $\min\{p_j - \tau,\, t_{k+1} - t_k\}$ time units.

(The value $t_{k+1} - t_k$ in the minimum expression is the length of the time slot and hence imposes a hard upper bound on the processing of any job piece during the slot.) It remains to fix the value of threshold τ. As the length of the processed job pieces $\min\{p_j - \tau,\, t_{k+1} - t_k\}$ decreases monotonically with τ and as we want to process the jobs as much as possible, we choose τ as the smallest non-negative real number which satisfies

$$\sum_{j:p_j>\tau} \min\{p_j - \tau,\, t_{k+1} - t_k\} \;\leq\; m_k \cdot (t_{k+1} - t_k). \tag{1}$$

The left-hand side of (1) denotes the total job length packed into the slot, and the right-hand side of (1) imposes the upper bound from Proposition 1.2.(ii). By Proposition 1.2 all selected job pieces can indeed be scheduled during the time slot $[t_k, t_{k+1}]$. This completes the description of algorithm FatMcN.

We conclude this section with some observations on the schedule produced by algorithm FatMcN that will be crucial in the analysis. First, we note that FatMcN maximizes the total length of skinny job pieces processed during slot

$[t_k, t_{k+1}]$. Secondly, the processing of jobs during slot $[t_k, t_{k+1}]$ maintains their relative ordering with respect to their lengths:

Lemma 3.1. *Let p_i and p_j be the remaining processing times of two jobs at time t_k, and let x_i^{on} and x_j^{on} be the amounts of processing that these jobs receive from algorithm* FatMcN *during slot $[t_k, t_{k+1}]$. If $p_i \leq p_j$, then $x_i^{on} \leq x_j^{on}$ and $p_i - x_i^{on} \leq p_j - x_j^{on}$.*

Correctness of the algorithm. We will now prove that algorithm FatMcN always minimizes the makespan and hence indeed is 1-competitive. To this end we fix an arbitrary instance and consider an optimal offline schedule S^* and the corresponding online schedule S^{on} for it. The following lemma follows by a switching argument which we omit.

Lemma 3.2. *W.l.o.g. we may assume that the optimal schedule S^* processes the fat jobs during the same time slots as schedule S^{on}.*

By Lemma 3.2 we will assume from now on that the two schedules S^* and S^{on} only differ in their handling of some skinny jobs and hence are governed by the same sequence of critical time points $t_0 < t_1 < \cdots < t_s$. Let $[t_k, t_{k+1}]$ be the earliest time slot during which schedules S^* and S^{on} disagree in processing the skinny jobs, so that at least one skinny job receives different amounts of processing in S^* and S^{on}. If schedule S^{on} processes x^{on} time units of some job during the slot while S^* processes x^* time units of the job, then we say that the two schedules have an overlap of $\min\{x^{\text{on}}, x^*\}$ with respect to this job. As a measure of progress we will use the sum of the overlaps taken over all jobs. We will show how to increase this total overlap by restructuring the optimal schedule S^*, without worsening its makespan.

First we observe that the total length of skinny jobs processed during slot $[t_k, t_{k+1}]$ in schedule S^{on} is at least as larger as in S^*: If $\tau > 0$ in the algorithm, then S^{on} has no idle time and S^* cannot fit more. If $\tau = 0$ then S^{on} schedules the maximal possible part of each skinny job and thus S^* cannot complete more, either.

First we claim that we can assume that the total length of skinny jobs processed during slot $[t_k, t_{k+1}]$ is the same in schedules S^{on} and S^*.

Next suppose that schedule S^{on} processes larger total length of skinny jobs than S^*, let the difference be z. It follows that we can move one or more skinny job pieces of a job with $x^* < x^{\text{on}}$ from some later time slot into $[t_k, t_{k+1}]$; we choose the total length of the pieces to be $\min\{x^{\text{on}} - x^*, z\}$. This increases the overlap and does not violate the conditions of Proposition 1.2, thus S^{on} may be rearranged in $[t_k, t_{k+1}]$ into a valid schedule. After a finitely many steps we have $z = 0$, as we are moving pieces of each job only once.

Now, in the remaining cases, schedules S^* and S^{on} both process exactly the same total length of skinny jobs during the slot. As the schedules differ, there exists a job J_i that during the slot receives more processing in S^* than in S^{on} and there exists another job J_j that receives more processing in S^{on} than in S^*.

If we denote the corresponding four job pieces by x_i^{on} and x_i^* (for job J_i) and by x_j^{on} and x_j^* (for job J_j), then this means

$$x_i^{\text{on}} < x_i^* \qquad \text{and} \qquad x_j^{\text{on}} > x_j^* \tag{2}$$

We denote the remaining processing time of jobs J_i and J_j at time point t_k by p_i and p_j. As the schedules S^* and S^{on} fully agree up to time t_k, these values are the same in both schedules. The following lemma follows from the properties of FatMcN.

Lemma 3.3. *The jobs J_i and J_j satisfy $p_j - x_j^* > p_i - x_i^*$.*

By Lemma 3.3 schedule S^* must contain a non-trivial time slot $[u, v]$ with $u \geq t_{k+1}$, during which job J_j is processed continuously while job J_i is not processed at all. We choose such an interval where u and v are preemption times of some jobs and let $\varepsilon = \min\{x_i^* - x_i^{\text{on}}, v - u\}$. We switch an ε-piece of job J_i from slot $[t_k, t_{k+1}]$ with an ε-piece of job J_j in slot $[u, v]$. While this keeps schedule S^* feasible and optimal, it also improves the total overlap.

We repeatedly perform such switches until eventually the overlap covers all the processing time in the slot, so that schedule S^* agrees with schedule S^{on} on time slot $[t_k, t_{k+1}]$. To see that the process is finite, note that by the choice of u and v there is only a fixed number of intervals $[u, v]$ we can use (the number is given by the number of preemptions in the schedule), thus after a fixed number of switches the schedules S^{on} and S^* must agree on an additional job and eventually on all jobs in the time slot. Then we handle the remaining time slots in the same fashion, and eventually transform the optimal schedule S^* into the online schedule S^{on} without ever worsening the makespan. Consequently the schedule S^{on} produced by algorithm FatMcN has the optimal makespan, which means that FatMcN is 1-competitive.

4 The negative result for fully online algorithms

In this section we show that if W violates one of the conditions (c1), (c2), (c3), then there is no 1-competitive *fully* online algorithm on m machines under the width set W. Throughout we may assume that W actually satisfies conditions (c1) and (c2), as otherwise the arguments in Section 2 apply and exclude the existence of a nearly and thus also of a fully online algorithm. Hence condition (c3) is violated, so that there exist $a \in W^-$ and $c \in W^+$ with $R(a, m-c) \geq 1$ and $R(a, m) \geq 3$. This condition implies that a job of width c may be replaced by two jobs of width a, or more precisely $R(a, m) \geq R(a, m-c)+2$. If $R(a, m-c) = 1$ then this follows from $R(a, m) \geq 3$. If $R(a, m-c) \geq 2$ then $c > m/2$ implies $R(a, c) \geq R(a, m-c) \geq 2$, which yields $R(a, m) \geq R(a, c) + R(a, m-c) \geq R(a, m-c) + 2$, and the condition holds as well.

Once again we use an adversarial argument. In all possible cases, the optimal makespan of the resulting job set will be 4. The first adversarial phase confronts the scheduler at time 0 with one fat job of width c and length 2, with two skinny

crucial jobs of width a and length 1, and with $R(a, m-c)-1$ skinny *dummy* jobs of width a and length 4. We stress that if $R(a, m-c) = 1$ then there are no dummy jobs. It is easily verified that the optimal offline makespan for this job set is at most 4.

Next the adversary spends some time waiting and observing the actions of the 1-competitive fully online scheduler. Let $t > 0$ be the first moment in time where the online algorithm changes the collection of running jobs (by preempting a job, or by completing a job, or by starting a new job on a previously idle machine).

- During the time interval $[0, t]$, the online scheduler must continuously process all the $R(a, m-c)-1$ dummy jobs. If $R(a, m-c) = 1$, this statement is trivial. If $R(a, m-c) \geq 2$ and no further jobs arrive, this is the only way to prevent the online makespan from exceeding the optimal makespan of 4.
- During $[0, t]$ the online scheduler must continuously process the fat job of length 2. Otherwise another fat job of width c and length 2 arrives at time 2. The optimal schedule has makespan 4, whereas the online schedule cannot complete both fat jobs by time 4.

As $c + a \cdot (R(a, m-c)-1) + 2a > m$, the online scheduler does not have sufficient space to process both crucial skinny jobs during the time interval $[0, t]$.

The second adversarial phase starts at time t, when a skinny job of width a and length $4-t$ arrives together with a fat job of width c and length $1+t/2$. The optimal offline schedule still has makespan 4. Indeed, the optimal schedule uses $a \cdot (R(a, m-c)-1)$ machines to continuously process the dummy jobs during $[0, 4]$. It uses a further machines to first process a piece of length $t/2$ of one crucial job, then a piece of length $t/2$ of the other crucial job, and finally the skinny job of length $4-t$ that arrives at time t. It uses c machines to first process the two fat jobs during $[0, 3+t/2]$ and then during $[3+t/2, 4]$ the remaining pieces of length $1-t/2$ of the two crucial jobs; this is feasible as $R(a, m) \geq R(a, m-c)+2$.

The online scheduler, however, must block $a \cdot R(a, m-c)$ machines from time t onwards just in order to complete the dummy jobs and the job of length $4-t$ that arrives at time t. This leaves a fat job of length 2, a fat job of length $1+t/2$, and a crucial job of length 1 that has not been processed at all before time t. These three jobs cannot be completed on the remaining machines by time 4. Hence the makespan produced by the fully online scheduler will be above 4, and a fully online scheduler cannot be 1-competitive.

5 The positive result for fully online algorithms

In this section we construct 1-competitive *fully* online scheduling algorithms for all the cases where the job widths in W satisfy conditions (c1), (c2), and (c3) in Theorem 1.1. We will separately discuss two scenarios. The first scenario has $R(a, m-c) = 0$ for all $a \in W^-$ and $c \in W^+$ in condition (c3). The second scenario has $R(a, m) = 2$ for some $a \in W^-$ in condition (c3).

The first scenario. If $R(a, m-c) = 0$ holds for all $a \in W^-$ and $c \in W^+$, then no fat job can be processed simultaneously with a skinny job. Furthermore any set of $R = R(a, m)$ skinny jobs can be processed on the machines in parallel.

We sketch an online algorithm for this scenario. Whenever there are fat jobs available, we run an arbitrary fat job. Similarly as in Lemma 3.2 it can be seen that there is an optimal schedule that handles the fat jobs in exactly the same way as our fully online algorithm. However, this time we are in a simpler situation as the processing of fat jobs and the processing of skinny jobs must occur in disjoint time slots, and thus cannot interfere with each other. Hence we may use the 1-competitive fully online algorithm of Hong & Leung [7] as stated in Proposition 1.3 to schedule the skinny jobs. All in all, this yields a fully online algorithm for the first scenario.

The second scenario. If $R(a, m) = 2$ holds for some $a \in W^-$, then conditions (c1) and (c2) imply that $R(a, m) = 2$ for all $a \in W^-$ and that the fat jobs can be divided into two types: *very fat jobs* which cannot be processed simultaneously with any skinny job and the remaining *fat jobs* which can be processed with one arbitrary skinny job.

The main idea of our fully online algorithm TwoFatMcN is first to handle the very fat and fat jobs with high preference and then to fit the skinny jobs into the remaining space. This is easier than for FatMcN as we combine at most two jobs and then the only obstacle against balancing them exactly is if one job is longer than the total processing time of the remaining jobs.

Let at any time $p_1 \geq p_2 \geq \cdots \geq p_s$ denote the processing times of the skinny job pieces that are available and still need processing. Let P denote their total remaining time, $P = \sum_{i=1}^{s} p_s$ and let R denote the total processing time of the fat (but not very fat) job pieces that are available and still need processing.

The algorithm TwoFatMcN at each *decision time* determines the schedule for some future interval. However, whenever a new job arrives, the schedule is stopped immediately and a new decision is made. Thus the next decision time is either the next arrival time or the time when the prespecified schedule ends.

(1) If a very fat job is available, run one such job until its completion.
(2) If a fat job is available, run the first such job f (chosen by some canonical ordering). Run also one skinny job (if available) chosen as follows:
   (a) If no skinny job is available, run only f until its completion.
   (b) If $p_1 > p_2$, run the job with remaining time p_1 for time $p_1 - p_2$ or until the completion of f, whatever happens first. Also, if there is a single skinny job, run it for time p_1 or until the completion of f, whatever happens first.
   (c) If $p_1 = p_2$, run the job with remaining time p_2 for time $\min\{p_2, R/2\}$ or until the completion of f, whatever happens first.
(3) If no fat and no very fat job is available:
   (a) If there is a single skinny job run it until its completion.
   (b) Otherwise, if $p_1 > P/2$, run the job with remaining time p_1 together with one other arbitrary job until the completion of the second job.
   (c) Otherwise, create a schedule of length $P/2$ for the skinny jobs using McNaughton's rule and follow it.

It is not immediately clear that the algorithm is finite, as in steps (2b) and (2c) no job may complete or arrive. However, note that while running a single

fat job f, each step (2b) is followed by step (2c). Furthermore, a job j can be run in step (2c) only once: The step (2c) takes time $R/2$ and after that, the other job with remaining time p_1 would need to run for $R/2$ before p_2 ties the longest remaining time again. However, this together would take time R which means that f is completed.

Correctness of the algorithm. We will now prove that algorithm TwoFatMcN always minimizes the makespan and hence indeed is 1-competitive. To this end we fix an arbitrary instance and consider an optimal offline schedule S^* and the corresponding online schedule S^{on} for it. The next lemma is proven by the same exchange argument as Lemma 3.2 We omit the proof.

Lemma 5.1. *W.l.o.g. we may assume that the optimal schedule S^* processes the fat and very fat jobs during the same time slots as schedule S^{on}.*

Let at any time $Z = \max\{R, p_1, (P+R)/2\}$, taking $p_1 = 0$ if no skinny job is available. Note that Z is the length of the optimal schedule for the remaining pieces of skinny and fat jobs if no further jobs arrive. Lemma 5.1 implies that at any time, the remaining pieces of fat jobs in S^* have total length R. Let P^*, p_1^*, and Z^* denote the values P, p_1, and Z with respect to the optimal schedule S^*. The following invariant follows inductively from the definition of the algorithm, details are omitted.

Lemma 5.2. *At any time during the execution of* TwoFatMcN *we have*

$$P \leq P^* \quad \textit{and} \quad Z \leq Z^*. \tag{3}$$

Lemma 5.1 and Lemma 5.2 together imply that as long as S^{on} has some unfinished job, also S^* has an unfinished job. Thus the makespan of S^{on} is equal to the makespan of S^{on} and TwoFatMcN is a fully online 1-competitive algorithm.

6 Conclusions

Now that we understand the 1-competitive cases of problem $P|\text{pmtn}, \text{size}_j, r_j|C_{\max}$, the next goal should be to get a better understanding of the competitive ratios in the remaining cases. Determining the best possible competitive ratio for every possible width set W and every possible number m of machines might be a messy and hopeless enterprise. A realistic first step could be to determine the best possible competitive ratio c^* for the general online problem $P|\text{pmtn}, \text{size}_j, r_j|C_{\max}$. Currently, we only know $6/5 \leq c^* \leq 2$ from the work of Johannes [8] and Naroska & Schwiegelshohn [11].

Our main Theorem 1.1 implies that for $m = 2$ and $m = 3$ machines there exist 1-competitive online algorithms. The smallest open problems arise on $m = 4$ machines. What is the optimal competitive ratio for $m = 4$ and $W = \{1, 2\}$? What is the optimal competitive ratio for $m = 4$ and $W = \{1, 2, 3\}$?

And what if we a priori know the optimal makespan? We have observed that the algorithm of Hong & Leung uses this knowledge but not in any significant

way. We know that knowing the optimal makespan cannot help us to design a 1-competitive algorithm: in all our constructions, we may as well announce the optimal makespan to be a fixed large value at the beginning and instead of ending the instance, we could release another batch of jobs that fully utilizes the machines till the announced optimal makespan. However, knowing the optimal makespan (intuitively speaking) should improve the competitive ratio.

Acknowledgements. We are grateful to Martin Böhm for the observation on the known optimum. Jiří Sgall acknowledges support by the project 14-10003S of GA ČR. Gerhard Woeginger acknowledges support by the Alexander von Humboldt Foundation, Bonn, Germany.

References

1. S. Albers and G. Schmidt (2001). Scheduling with unexpected machine breakdowns. *Discrete Applied Mathematics 110*, 85–99.
2. J. Blazewicz, M. Drabowski, and J. Weglarz (1986). Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers 35*, 389–393.
3. M. Drozdowski (1992). Problems and algorithms of multiprocessor tasks scheduling. PhD thesis, Technical University of Poznan, Department of Computer Science.
4. M. Drozdowski (1995). On complexity of multiprocessor task scheduling. *Bulletin of the Polish Academy of Sciences, Technical Sciences, 43*, 381–393.
5. M. Drozdowski (1996). Scheduling multiprocessor tasks — an overview. *European Journal of Operational Research 94*, 215–230.
6. S. Guo and L. Kang (2013). Online scheduling of parallel jobs with preemption on two identical machines. *Operations Research Letters 41*, 207–209.
7. K.S. Hong and J.Y.-T. Leung (1992). On-line scheduling of real-time tasks. *IEEE Transactions on Computers 41*, 1326–1331.
8. B. Johannes (2006). Scheduling parallel jobs to minimize the makespan. *Journal of Scheduling 9*, 433–452.
9. J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan (1984). Preemptive scheduling of uniform machines subject to release dates. In: *Progress in Combinatorial Optimization*, W.R. Pulleyblank (Ed.), Academic Press, 245–261.
10. R. McNaughton (1959). Scheduling with deadlines and loss functions. *Management Science 6*, 1–12.
11. E. Naroska and U. Schwiegelshohn (2002). On an on-line scheduling problem for parallel jobs. *Information Processing Letters 81*, 297–304.
12. K. Pruhs, J. Sgall, and E. Torng (2004). Online scheduling. In: *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, J.Y.T. Leung (Ed.), Chapman & Hall/CRC, chapter 15, 1–41.
13. G. Schmidt (1984). Scheduling on semi-identical processors. *Zeitschrift für Operations Research*, 153–162.
14. J. Sgall (1998). On-line scheduling. In: *Online Algorithms: The State of the Art*, A. Fiat and G.J. Woeginger (Eds.), Lecture Notes in Computer Science 1442, Springer, 196–231.
15. A.P.A. Vestjens (1998). Scheduling uniform machines on-line requires nondecreasing speed ratios. *Mathematical Programming 82*, 225–234.

A Proof of Lemma 2.1

Consider the auxiliary function $f(k) = R(a, m - kb)$, for $k = 0, \ldots, r$, intuitively the number of jobs of width a that can be scheduled together with k jobs of width b. Also $f(k+1) < f(k)$ for all $k = 1, \ldots, r-1$, as $a < b$ and thus if $f(k+1)$ items of width a fit with $k+1$ items of with b, then $f(k)+1$ items of width a fit with k items of with b.

Now, for the sake of contradiction, assume that $x_0 = 0$ and $x_1 = 1$. Since $x_0 = 0$, the maximal utilization configuration contains only items of width b and thus $y_0 = r$ and $f(r) = 0$. Similarly, $x_1 = 1$ implies $y_1 = r - 1$ and $f(r-1) = 1$. This implies that the number of utilized machines is $(r-1)b + a = rb + a - b \leq m + a - b$, as $rb \leq m$.

Furthermore, since $(r-1)b + a$ maximizes the utilization for confiburations containing a, we have for all $k = 0, \ldots, r-2$ that $kb + f(k)a \leq (r-1)b + a \leq m + a - b$. This implies that in this configuration we can replace one job of width a by a job of width b without exceeding the bound of m. Thus $f(k+1) \geq f(k) - 1$. Together with $f(k+1) < f(k)$ we have $f(k+1) = f(k) + 1$ and thus from $f(r-1) = 1$ we get $f(0) = r$. However obviously $f(0) = r(a)$ and thus $r(a) = r = r(b)$, contradicting the assumption of the lemma.

B Remaining cases from Section 2

B.1 Analysis of the second case

In this section we assume that condition (c2) is violated for $a, b \in W^-$ and $c \in W^+$ with $R(a, m-c) > R(b, m-c) \geq 2$. We also assume that condition (c1) holds with $R(a, m) = R(b, m)$, as otherwise we could proceed as in Section **??**.

The argument for this case is a small modification of the argument in Section **??**. We essentially recycle our old adversarial instance for $m' = m - c$ machines and widths a and b with $R(a, m') > R(b, m') \geq 2$. Additionally to the jobs in the old instance, at the beginning of the first phase there arrives a fat job of width c and length r, and at the beginning of the second phase there arrives another fat job of width c and length $r - 1 + L$; here r and L are defined as in Section **??** (for widths a and b and for m' machines). These two fat jobs continuously block c of the m machines from time 0 to time $r - 1 + L$, both in the optimal schedule and in the schedule of any 1-competitive nearly online algorithm. Hence the real battlefield are the remaining $m' = m - c$ machines, on which the online algorithm is beaten in exactly the same fashion as in Section **??**.

B.2 Analysis of the third case

Throughout this section we assume that condition (c2) is violated for $a, b \in W^-$ and $c \in W^+$ with $R(a, m-c) > R(b, m-c)$ and $R(b, m-c) \in \{0, 1\}$. We also assume that condition (c1) holds with $R(a, m) = R(b, m)$; note that this implies $a < b < 2a$ by Observation 1.4.

Lemma B.1. *Let x_2 and y_2 be integers that maximize the value of $ax + by$ subject to the constraints $ax + by + c \leq m$ and $x, y \geq 0$. Then $x_2 \geq 1$.*

Proof. We distinguish three cases. First if $R(b, m-c) = 0$, then $x_2 = R(a, m-c) \geq 1$ and $y_2 = 0$. Secondly if $R(b, m-c) = 1$ and $a+b+c \leq m$, then either $y_2 = 1$ and $x_2 \geq 1$, or $y_2 = 0$ and $x_2 = R(a, m-c) \geq 2$. In the third case $R(b, m-c) = 1$ and $a+b+c > m$. Then $R(a, m-c) \geq 2$ and $b < 2a$ imply $y_2 = 0$ and $x_2 = R(a, m-c) \geq 2$. □

The adversarial argument uses the values $x_2 \geq 1$ and $y_2 \geq 0$ introduced in Lemma B.1. We define $r = R(b, m - ax_2 - by_2)$. We observe that $ax_2 + by_2 + c \leq m$ and $b < c$ imply $R(b, m) > r \geq 1$. The first adversarial phase starts at time 0, and confronts the online scheduler with x_2 jobs of width a and length $R(b, m)$, with y_2 jobs of width b and length $R(b, m)$, and with $r \cdot R(b, m)$ jobs of width b and length 1. The second adversarial phase starts at time r, when a single fat job of width c and length $R(b, m)$ arrives.

The optimal schedule S_1 for the first phase continuously processes the $x_2 + y_2$ jobs of length $R(b, m)$ together with some r of the unit length jobs, and thus has makespan $R(b, m)$. The optimal schedule S_2 for the full instance processes all the unit length jobs of width b during the interval $[0, r]$, and then processes all the jobs of length $R(b, m)$ during the interval $[r, r + R(b, m)]$. This schedule utilizes $b \cdot R(b, m)$ machines during $[0, r]$ and $ax_2 + by_2 + c$ machines during $[r, r + R(b, m)]$.

Any 1-competitive online algorithm must follow the structure of the optimal schedule S_1 up to time r, and thus utilizes only $ax_2 + b \cdot R(b, m - ax_2)$ machines during $[0, r]$. As $x_2 \geq 1$ by Lemma B.1 and as $R(b, m) = R(b, m - ax_2) + x_2$ by Observation 1.4, this utilization is weaker than the one in the optimal schedule S_2. During the interval $[r, r + R(b, m)]$, the online algorithm must continuously run the job of width c. By the choice of x_2 and y_2 in Lemma B.1, the resulting utilization cannot beat the utilization in schedule S_2 during this interval. Summarizing, the utilization in the optimal schedule beats the online utilization and the optimal makespan beats the online makespan.

C Proof of Lemma 3.2

Let t be the earliest moment in time where S^* and S^{on} disagree in processing the fat jobs. Then during some non-trivial time interval $[t, t + \varepsilon]$, schedule S^{on} runs a fat job J' while the optimal schedule S^* either runs another fat job J'' with $w(J'') \leq w(J')$ or does not run any fat job at all. Let $u > t$ be the earliest moment in time where S^* runs the job J'. By choosing ε appropriately, we may assume that S^* runs J' all through $[u, u + \varepsilon]$.

Let us switch the two slices $[t, t + \varepsilon]$ and $[u, u + \varepsilon]$ of the optimal schedule S^*. This switch might cause some skinny job to be processed before its release date, but this infeasibility is easily repaired by appropriately switching skinny job pieces between the two slices; note that the slice $[t, t+\varepsilon]$ has at least as much space for skinny jobs as slice $[u, u + \varepsilon]$. By repeatedly switching such slices, we eventually establish the statement of the lemma. □

D Proof of Lemma 3.3

Suppose for the sake of contradition that $p_j - x_j^* \leq p_i - x_i^*$. By combining this with the inequalities in (2), we get

$$p_j - x_j^{\text{on}} \;<\; p_j - x_j^* \;\leq\; p_i - x_i^* \;<\; p_i - x_i^{\text{on}}. \tag{4}$$

Now (4) and (the contrapositive of) Lemma 3.1 imply $p_j < p_i$. Using $p_j < p_i$ and again Lemma 3.1 with j and i exchanged, we obtain $x_j^{\text{on}} \leq x_i^{\text{on}}$. Now the trivial bounds $x_j^* \geq 0$ and $x_i^* \leq t_{k+1} - t_k$ yield the chain of inequalities

$$0 \;\leq\; x_j^* \;<\; x_j^{\text{on}} \;\leq\; x_i^{\text{on}} \;<\; x_i^* \;\leq\; t_{k+1} - t_k. \tag{5}$$

By the behavior of algorithm FatMcN this means $x_j^{\text{on}} = p_j - \tau$ and $x_i^{\text{on}} = p_i - \tau$, where τ is the threshold value for slot $[t_k, t_{k+1}]$. But then $p_j - x_j^{\text{on}} = p_i - x_i^{\text{on}} = \tau$, which blatantly contradicts (4). This contradiction establishes the lemma. □

E Proof of Lemma 5.2

Proof. If some job arrives, both P and P^* increase by the same amount, thus $P \leq P^*$ is maintained. This also implies that $Z \leq Z^*$ is maintained unless $Z = p_1$ and p_1 increases. If p_1 increases then p_1 is the processing time of the newly arrived job; then $p_1 \leq Z^*$ and (3) is maintained as well.

To analyze the invariant during the steps of the algorithm, suppose the step takes time t between two decision points. During that time if no fat job is running, P^* can decrease by at most $2t$, and if a fat job is running then both R and P^* decrease by at most t. Thus in each case $(P^* + R)/2$ decreases at most by t. Also p_1^* trivially decreases by at most t. Thus also Z^* decreases by at most t. To prove that (3) is maintained, it is sufficient to show that P and Z decrease (at least) by the same amount.

- If in step (1) none of P, p_1, R and Z change, Lemma 5.1 implies that the same holds for S^*, and (3) is maintained.
- During the entire step (2a), $P = 0$ and $Z = R$, thus Z decreases by t and (3) is maintained.
- In step (2b), all values P, p_1, R decrease by t, so that also Z decreases by t. Since S^* is also running a fat job, (3) is maintained.
- In step (2c), P and R again decrease by t. Let now Z, P and R refer to the values at the beginning of the step. We claim that $Z - t \geq p_1$ and thus Z decreases by t even though p_1 does not change. The claim follows since $t \leq R/2$ and $P \geq 2p_1$ by the definition of the step, so that $Z - t \geq (P + R)/2 - t \geq P/2 \geq p_1$. Thus P and Z decrease at least as much as P^* and Z^*, and (3) is maintained.
- During the entire step (3a), $R = 0$ and $P = p_1 = Z$. Thus P and Z decrease by t. Since Z^* cannot decrease faster, $Z \leq Z^*$ is maintained. For $P \leq P^*$, it is sufficient to notice that at the end of the step, since $R = 0$, we have $Z^* = \max\{p_1^*, P^*/2\} \leq P^*$ and thus $P = Z \leq Z^* \leq P^*$.

– In step (3b), $R = 0$, P decreases by $2t$, and p_1 decreases by t, so that Z decreases by t and (3) is maintained.
– In step (3c), $R = 0$, P decreases by $2t$ and from the properties of the McNaughton's optimal schedule it follows that at all times $Z = P/2$. Thus Z decreases by t, and (3) is maintained.

Summarizing, we have shown that invariant (3) indeed holds at any time. □