

Fairness in periodic real-time scheduling*

Sanjoy K. Baruah

New Jersey Institute of Technology

Abstract

The issue of temporal fairness in periodic real-time scheduling is considered. It is argued that such fairness is often a desirable characteristic in real-time schedules. A concrete criterion for temporal fairness — pfairness — is described. The weight-monotonic scheduling algorithm, a static priority scheduling algorithm for generating pfair schedules, is presented and proven correct. A feasibility test is presented which, if satisfied by a system of periodic tasks, ensures that the weight-monotonic scheduling algorithm will schedule the system in a pfair manner.

Keywords: static-priority uniprocessor scheduling, periodic task systems, temporal fairness, weight-monotonic scheduling algorithm.

1 Introduction

A **periodic task** x is characterized by a *period* $x.p \in \mathbf{N}$ and a *computation requirement* $x.e \in \mathbf{N}$, with the interpretation that the task x expects to be allocated the processor for $x.e$ units of time in every interval $\{t | i \cdot x.p \leq t < (i+1) \cdot x.p\}$, for each $i \in \mathbf{N}$ (where \mathbf{N} denotes the set of natural numbers $0, 1, 2, \dots$). Given an instance Φ of n such periodic tasks, the (uniprocessor) **periodic scheduling problem** [9] is concerned with attempting to schedule these n tasks on a single processor so as to satisfy the constraints of each task. Task preemption is permitted, but only at integral boundaries (i.e., the processor is allocated to a particular task, or remains unallocated, in integer multiples of the basic time unit).

For a given feasible set of tasks Φ , there are in general several valid schedules. For example, consider an instance Φ with two tasks x and y , with $x.e = 3, x.p = 5, y.e = 2, y.p = 5$. A valid schedule $S1$ may allocate the processor to x at time-slots $\{t | t \equiv 0 \pmod{10} \text{ or } t \equiv 1 \pmod{10} \text{ or } t \equiv 2 \pmod{10} \text{ or } t \equiv 7 \pmod{10} \text{ or } t \equiv 8 \pmod{10} \text{ or } t \equiv 9 \pmod{10}\}$ (we number time-slots starting with 0 — see Figure 1),

and to task y at all remaining time slots; another, $S2$, may allocate to x at all time-slots $\{t | t \equiv 0 \pmod{5} \text{ or } t \equiv 1 \pmod{5} \text{ or } t \equiv 3 \pmod{5}\}$, and to y the rest of the time. While schedule $S1$ has the advantage of fewer preemptions, schedule $S2$, “spreads out” the allocations to both tasks, providing both with less bursty service (in $S1$ task y gets 4 consecutive slots and is then starved for 6 slots, while, in $S2$, successive allocations to y are separated by at least one, and at most two, slots.) Which is better depends, of course, upon the application — if preemptions are expensive, $S1$ would be preferred; if, on the other hand, the attempt is to minimize the length of time during which a task is denied service, then $S2$ is the schedule of choice, since it is, in an intuitive sense, “fairer.”

Temporal fairness. This issue of *fairness* in resource-allocation and scheduling has recently been attracting considerable attention [1, 2, 3, 4]. Motivated no doubt in part by applications, such as multimedia, which are characterized by fairly “regular” resource requirements over extended intervals, attempts have been made to formalize and characterize notions of temporal fairness. This research addresses the issue of designing fair schedules for systems of periodic tasks. The notions of *proportionate progress* and *pfairness* introduced in [4] (see also [5]) are used as a measure of the fairness of a schedule. Informally (these concepts will be made more precise in the following sections), a schedule displays proportionate progress (equivalently, satisfies *pfairness*, or is *pfair*) if at all integer time instants t and for all tasks x , the **lag** of task x at time t — the difference between the amount of time for which x should have been allocated a processor, and the amount of time for which it was allocated a processor — is strictly less than 1 in absolute value. It was proven in [4] that pfair scheduling is a stronger requirement than periodic scheduling, in that any pfair schedule is periodic. The converse, however, is not generally true — periodic schedule $S1$ above is not pfair.

It has been shown [9] that $\sum_{x \in \Phi} (x.e/x.p) \leq 1$ is a necessary and sufficient condition for a system Φ of pe-

*This research has been supported in part by NSF grants OSR-9350540 and CCR-9410752, and UVM grant PSC194-3.

riodic tasks to have a periodic schedule; furthermore, the *earliest deadline first* scheduling algorithm [6] has been proven an optimal scheduling algorithm. Somewhat surprisingly, the condition $\sum_{x \in \Phi} (x.e/x.p) \leq 1$ has also been shown to be (necessary and) sufficient for pfair feasibility: *Algorithm PF* [4] is an optimal scheduling algorithm that generates pfair schedules.

On-line scheduling algorithms. System designers who desire to construct schedules for periodic task systems have the option of precomputing a schedule by simulating the behavior of the earliest deadline first algorithm (or Algorithm PF, if a pfair schedule is desired) beforehand, and then implementing the scheduler by a table lookup during system execution. The correctness of this method follows from the observation that an infinite schedule for task system Φ can be obtained by infinitely repeating a (correct) schedule of length $\text{lcm}_{x \in \Phi} \{x.p\}$; however, the arguments against such an approach are many, the major ones being that, in general, exponential time is required to precompute the schedule and exponential space required to store it. A more realistic approach is to implement the scheduler as follows: at each time slot, assign a **priority** to each contending¹ task, and allocate the processor for that time slot to the highest-priority task. Such schedulers have been referred to in the literature as *on-line* schedulers (although there is, in fact, nothing “on-line” about them: all relevant information — all the tasks’ computation requirements and periods — is available beforehand, and cannot change). Such an on-line implementation for the earliest deadline first algorithm would assign, at time t , a priority of $(t \text{ div } x.p + 1) \cdot x.p$ to each contending task x , with smaller values having higher priority than larger ones (and ties broken arbitrarily).

Dynamic and static priorities. With respect to the earliest deadline first algorithm, it is possible that some tasks x and y are both contending at times t and t' such that at time t , x has higher priority than y while at time t' , y has higher priority than x . Algorithms that permit such “switching” of priorities between tasks are known as *dynamic* priority algorithms.

¹ Informally, a task is *contending* at a given time-instant if it can be allocated the processor without violating any constraints. If a periodic schedule is desired, this requires that the task not have received its computation requirement for the current period; if pfairness is the aim, then it is necessary that allocating the processor to this task for another time unit not drive its lag below -1. These concepts will be formally defined in the sections that follow.

By contrast, *static* priority algorithms satisfy the property that for every pair of tasks x and y , whenever x and y are both contending, it is always the case that the same task has priority. An example of a static-priority scheduling algorithm for periodic scheduling is the rate-monotonic scheduling algorithm [9], which assigns a priority $x.p$ to task x , with smaller values having priority over larger ones and ties broken arbitrarily, but in a consistent manner: if $x.p = y.p$ and x is given priority over y once, then it is always given priority over y .

Whether dynamic-priority based scheduling algorithms are preferred in a particular situation, or static-priority based ones, depends upon a variety of factors. Dynamic priority algorithms can sometimes schedule problem instances that cannot be scheduled by any static priority algorithm (with respect to the Periodic Scheduling Problem, instance $\Phi = \{x, y\}$, with $x.e = 5, x.p = 10, y.e = 11, y.p = 25$, with $\sum_{x \in \Phi} (x.e/x.p) = 0.94$, is an example); by contrast, any instance that can be scheduled by a static scheduling algorithm can be scheduled by a dynamic scheduling algorithm (this statement is trivially true, since any static scheduling algorithm is also a dynamic scheduling algorithm). If implemented in software, execution times are often comparable — using, for example, the binomial heaps data structure [12], both the earliest deadline first and the rate-monotonic algorithms can be implemented to execute in $O(\log n)$ time per slot, where n is the number of periodic tasks in the instance.

There are certain circumstances where static-priority schedulers are, however, more appropriate:

Hardware implementation of priorities. Particularly when the number of tasks n is small, static task priorities are conveniently implemented by hardware interrupts. Each task has control over an interrupt pin of the processor, with higher-priority tasks having control over higher-priority pins. The decision on whether it is in a contending state or not is local to a task. If contending, a task activates its interrupt line, and the highest priority contending task wrests control of the processor in essentially constant time. (Of course, such an implementation does not scale with problem size — special purpose hardware, with the number of interrupt levels proportional to the number of tasks — is needed.)

Distributed communicating tasks. Consider a number of tasks communicating via a shared medium such as a bus. Use of the shared medium

goes through a period of contention resolution, at the end of which one particular task is given exclusive use of the medium, followed by a period of data transmission by the successful task. This is followed by another contention period, another period of data transmission, and so on. Under heavy loads, “collision-free” contention resolution protocols such as the *binary countdown protocol* (see, e.g., [11]) guarantee higher bandwidth utilization, and are consequently preferred.

In the binary countdown protocol (and similar protocols), the tasks exchange information about their relative priorities by means of a clever scheme that takes time proportional to the binary logarithm of the largest priority value (for the earliest deadline first algorithm, e.g., this would be the furthest deadline among the contending tasks). In dynamic-priority algorithms, this largest value may be large, is often difficult to compute beforehand, and depends upon the individual characteristics of the tasks. For static-priority schemes, this largest value is the number of tasks n ; contention resolution consequently takes time proportional to $\log n$. In static-priority based schemes, therefore, the contention resolution interval is small in size (thus saving bandwidth); more important, this size is *a priori* predictable, thus permitting better static analysis of the system’s behavior.

This research. We consider here the problem of fair scheduling of systems of periodic tasks on a single processor. In Section 2, we define precisely the concept of pfair scheduling, and summarize the known results with respect to dynamic-priority pfair scheduling. These include a (necessary and sufficient) feasibility test, and an efficient scheduling algorithm. In Section 3 we discuss static-priority pfair scheduling. We present Algorithm WM — an efficient static priority scheduling algorithm for generating pfair schedules — and a feasibility test which, if satisfied by a problem instance Φ , ensures that Algorithm WM will schedule Φ in a pfair manner.

2 Pfairness: Definitions, and related work

We start with some conventions:

- We adopt the standard notation of having $[a, b)$ denote the contiguous natural numbers $a, a + 1, \dots, b - 1$.

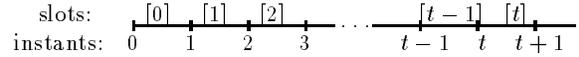


Figure 1: Notation: Time instants and time slots

- Scheduling decisions are made at integral values of time, numbered from 0. The real interval between time t and time $t+1$ (including t , excluding $t+1$) will be referred to as slot t , $t \in \mathbf{N}$ (see Figure 1).
- We will consider an instance that involves one processor and a set Φ of n tasks. Specific tasks will be denoted by (perhaps subscripted) identifiers x, y which range over the tasks in Φ .
- Each task x has two integer attributes — a *period* $x.p$ and an *execution requirement* $x.e$. We define the *weight* $x.w$ of task x to be the ratio $x.e/x.p$. Furthermore, we assume $0 < x.w \leq 1$.
- The quantity $(\sum_{x \in \Phi} x.w)$ is referred to as the *density* of instance Φ , and will be denoted by ρ .

Now some definitions:

- A *schedule* S for instance Φ is a function from the natural numbers to $\Phi \cup \{\perp\}$, with the interpretation that $S(t) = x$, $x \in \Phi$, if the processor is allocated to task x for slot t , and $S(t) = \perp$ if the processor is unallocated during time-slot t .
- Schedule S is a *periodic schedule* if and only if

$$\forall i, x : i \in \mathbf{N}, x \in \Phi : |\{t | t \in [0, x.p \cdot i) \text{ and } S(t) = x\}| = x.e \cdot i.$$

That is, each task x is allocated exactly $i \cdot x.e$ slots during its first i periods, for all i .

- The *lag* of a user x at time t with respect to schedule S , denoted $\text{lag}(S, x, t)$, is defined by:

$$\text{lag}(S, x, t) = x.w \cdot t - |\{t' | t' \in [0, t) \text{ and } S(t') = x\}|.$$

The quantity $x.w \cdot t$ represents the amount of time for which task x *should* have been allocated the processor over $[0, t)$, and $|\{t' | t' \in [0, t) \text{ and } S(t') = x\}|$ is equal to the number of slots for which task x was actually scheduled.

- A schedule S is *pfair* if and only if

$$\forall x, t : x \in \Phi, t \in \mathbf{N} : -1 < \text{lag}(S, x, t) < 1.$$

That is, a schedule is pfair if and only if it is never the case that any task x is overallocated or underallocated by an entire slot.

Pfairness is an extremely stringent form of fairness — indeed, it has been shown [4] that no stronger fairness can be guaranteed achievable for periodic task systems in general. (Consider a system of n identical tasks, each with weight $1/n$. The task that is scheduled at slot 0 has a lag $(-1 + 1/n)$ at time 1, and the one scheduled at slot $n - 1$ has a lag $(1 - 1/n)$ at time $(n - 1)$. By making n large, these lags can be made arbitrarily close to -1 and $+1$, respectively.)

Periodic schedules can also be defined in terms of lag constraints. In particular, a schedule S is periodic if and only if

$$\forall i, x : i \in \mathbf{N}, x \in \Phi : \text{lag}(S, x, x.p \cdot i) = 0,$$

from which it follows that *every pfair schedule is periodic*. (This follows from the observations that in the definition of lag, the term $x.w \cdot t$ is independent of S , and that the term $|\{t' | t' \in [0, t) \text{ and } S(t') = x\}|$ is an integer.)

With respect to a given task x , let $\text{earliest}(x, j)$ (resp., $\text{latest}(x, j)$) denote the earliest (resp., latest) slot during which x may be scheduled for the j th time, $j \in \mathbf{N}$, in any pfair schedule. We can easily derive closed-form expressions for $\text{earliest}(x, j)$ and $\text{latest}(x, j)$:

$$\begin{aligned} \text{earliest}(x, j) &= \min t : t \in \mathbf{N} : x.w \cdot (t + 1) - (j + 1) > -1 \\ &= \lfloor \frac{j}{x.w} \rfloor \end{aligned}$$

Similarly,

$$\begin{aligned} \text{latest}(x, j) &= \max t : t \in \mathbf{N} : x.w \cdot t - j < 1 \\ &= \lceil \frac{j+1}{x.w} \rceil - 1 \end{aligned}$$

A slot t such that $t = \text{earliest}(x, j)$ for some j is called an **activation-slot** for task x .

With respect to a particular schedule S ,

- $\text{allocated}_x(t) \stackrel{\text{def}}{=} |\{t' | t' \in [0, t) \text{ and } S(t') = x\}|$.
- Task x is **contending** at time t if it may receive the processor without becoming overallocated; i.e., if

$$\text{allocated}_x(t) = k \wedge \text{earliest}(x, k + 1) \leq t$$

This is an important difference between periodic and pfair schedules. In a periodic schedule, task x is contending at time t if $\text{allocated}_x(t) = k \wedge (k \text{ div } x.e) \cdot x.p \leq t$.

- For contending tasks, a **pseudo-deadline** is defined: the pseudo-deadline of task x is the time by which x must be allocated the processor if it is to not violate its lag constraints. More formally, for contending task x at time t , the **pseudo-deadline**

$$x.d \stackrel{\text{def}}{=} \text{latest}(x, \text{allocated}_x(t) + 1)$$

2.1 Dynamic-priority pfair scheduling

The concept of pfairness was introduced in [4], in the context of constructing periodic schedules for a system of periodic tasks on several identical processors — the *multiprocessor periodic scheduling problem* [8]. The following theorem was proved there, by means of some fairly involved network-flow constructions, and by using the Integer Flow Theorem [7]:

Theorem 1 *A system of periodic tasks can be scheduled in a pfair manner on m processors provided the weights of all the tasks sum to at most m .*

As a special case, we obtain the following corollary with respect to uniprocessor systems:

Corollary 2 *Every uniprocessor system of periodic tasks Φ for which $(\sum_{x \in \Phi} x.w \leq 1)$ holds has a pfair schedule.*

In addition, an on-line scheduling algorithm — Algorithm PF — was presented and proven correct. This algorithm has a non-trivial priority scheme that requires $O(\sum_{\text{all } x} \lceil \log(x.p + 1) \rceil)$ time to determine the m highest-priority tasks in the worst case. However, for the uniprocessor case, Algorithm PF reduces to simply allocating the processor at each time slot to the highest-priority contending task according to the following pseudo-deadline based priority rule:

contending task x has priority over contending task y iff $x.d \leq y.d$ — ties broken arbitrarily.

Again with respect to the uniprocessor case, Algorithm PF can be implemented using the heap-of-heaps data structure [10] in $O(\log n)$ time per time slot, where n is the number of tasks.

3 Static pfair scheduling

As we observed in the preceding section, the general problem of generating pfair schedules for systems of periodic tasks on a single processor has been pretty completely solved. A necessary and sufficient condition for feasibility has been obtained, and an efficient scheduling algorithm designed. We now turn our attention to a more restricted problem: the design of static-priority scheduling algorithms (and associated feasibility tests) that schedule systems of periodic tasks in a pfair manner.

3.1 Algorithm WM

Scheduling algorithm WM (for “weight-monotonic”) assigns the processor at each time slot to the contending task with the greatest priority according to the following – static – priority rule:

task x is assigned (static) priority over task y if $x.w \geq y.w$ — ties broken arbitrarily.

(Recall that, in the context of pfair scheduling, a task x is *contending* at time t if it may receive the processor without becoming overallocated; i.e., if $\text{allocated}_x(t) = k \wedge \text{earliest}(x, k+1) \leq t$.)

While Algorithm WM, at first glance, looks remarkably like the standard rate-monotonic scheduling algorithm [9], there is an important difference. While priorities in the rate-monotonic scheduling algorithm are assigned according to tasks’ periods, the sole criterion determining a task’s priority for Algorithm WM is its *weight*. Thus, a task x with $x.e = 1, x.p = 3$ may have lower priority than a task y with $y.e = 200, y.p = 300$. Intuitively, this seems reasonable from the point of view of fairness — both tasks require the processor two-thirds of the time. This priority scheme considers the task y to be equivalent to a task y' with $y'.e = 2$ and $y'.p = 3$, and thus avoids starving y for extended periods of time.

A question that arises at this point is: does this actually cost us anything? Specifically, does the fact that we have taken a task of low priority under the rate-monotonic scheme (task y , above) and elevated its priority make it less likely that a task system can be scheduled by Algorithm WM than by the rate-monotonic algorithm? Corollary 4 provides a partial answer to this question, in proving that the density-bound for feasibility by Algorithm WM is no worse than for rate-monotonic feasibility. Indeed, as the following example illustrates, our experiments reveal that it is often the case that task systems which the rate-monotonic algorithm fails to schedule are successfully scheduled by Algorithm WM.

Example 1 Let $\Phi = \{x, y\}$, with $x.e = 5, x.p = 10, y.e = 11, y.p = 25$. The rate-monotonic scheduling algorithm fails to generate a periodic schedule for Φ ; by the optimality of rate-monotonic scheduling [9], it follows that no static-priority scheduling algorithm can generate a periodic schedule for Φ . However, Algorithm WM successfully schedules Φ in a pfair manner. Since all pfair schedules are periodic, this implies that Algorithm WM generates a periodic schedule for Φ .

□

Example 1 presented an instance which could not be periodically scheduled by the rate-monotonic scheduling algorithm and hence, by any static-priority algorithm, but could be pfair scheduled (and thus, periodically scheduled) by the static-priority Algorithm WM. The apparent contradiction here is explained by the differences in the definitions of “contending” tasks in the contexts of periodic and pfair scheduling. Informally speaking, tasks tend to toggle between contending and non-contending status more often in pfair schedules than in periodic ones; *high-priority tasks therefore tend to grab the processor less completely in pfair schedules, and lower-priority ones are consequently less likely to be starved.*

A natural question to ask at this stage may be: is Algorithm WM able to schedule any task system Φ satisfying $\sum_{x \in \Phi} x.w \leq 1$ in a pfair manner? The answer, unfortunately, is “no”.

Example 2 The instance $\Phi = \{x, y, z\}$ with $x.w = 2/3, y.w = 1/5, z.w = 2/15$ has $\sum_{x \in \Phi} x.w = 1$, but cannot be scheduled by Algorithm WM.

□

We now present a density-based sufficient feasibility test for Algorithm WM. By *density-based test*, we mean that the sole property of task system Φ that determines whether Φ passes this test is its density — i.e., $\sum_{x \in \Phi} x.w$. A *sufficient* feasibility test is one such that, if an instance passes this test, then it is guaranteed that the algorithm under discussion will successfully schedule the instance.

Theorem 3 *Let Φ contain n tasks. If*

$$\sum_{x \in \Phi} x.w \leq \sum_{i=n}^{2n-1} \frac{1}{i}$$

then Φ can be scheduled in a pfair manner by Algorithm WM.

Proof: In the Appendix. \square

Since the density of instance Φ can be computed in $O(n)$ time, where n is the number of tasks, and the right-hand side of the inequality is the summation of n terms, this theorem immediately yields a linear-time feasibility test for Algorithm WM.

We observe, however, that the density condition of Theorem 3 above, while sufficient for feasibility, is not necessary. That is, there are instances of task systems which are scheduled in a pfair manner by Algorithm WM, but fail the test of Theorem 3. We present one such instance below:

Example 3 Consider an instance Φ with two tasks x and y , with $y.e = 37, y.p = 50, x.e = 13, x.p = 50$. $\sum_{x \in \Phi} x.w = 37/50 + 13/50 = 1.0$, while $\sum_{i=2}^3 1/i = 1/2 + 1/3 = 5/6$; instance Φ therefore fails the test of Theorem 3. Task system Φ is, however, scheduled in a pfair manner by Algorithm WM.

\square

The density bound for the rate-monotonic scheduling algorithm for a system of n tasks is $n(2^{\frac{1}{n}} - 1)$ [9, Theorem 5]. It may be instructive to compare this bound with the corresponding bound for Algorithm WM(Theorem 3):

Size of taskset	rate-monotonic bound	Algorithm WM's bound
2	0.828427	0.833333
3	0.779763	0.783333
4	0.756828	0.759524
5	0.743492	0.745635
10	0.717735	0.718771
20	0.705298	0.705803
50	0.697974	0.698172
100	0.695555	0.695653

Observe that the bound for Algorithm WM is always larger than the one for rate-monotonic scheduling in the table above. This is a consequence of the following algebraic identity (whose proof we omit):

$$\left(\forall n : n \geq 2 : n(2^{\frac{1}{n}} - 1) < \sum_{i=n}^{2n-1} \frac{1}{i} \right).$$

Furthermore, the bounds for both algorithms seem to converge with increasing task-set size. This is indeed the case, and follows from the observations that

$$\lim_{n \rightarrow \infty} \sum_{i=n}^{2n-1} \frac{1}{i} = \ln 2,$$

and

$$\lim_{n \rightarrow \infty} n(2^{\frac{1}{n}} - 1) = \ln 2,$$

where $\ln 2$ denotes the natural logarithm of 2.

Corollary 4 now follows from Theorem 3:

Corollary 4 Any task system Φ satisfying $(\sum_{x \in \Phi} x.w) \leq \ln 2$ can be scheduled in a pfair manner by Algorithm WM.

4 Conclusions

While the Periodic Scheduling Problem (PSP) has been extensively studied, it is only recently that the issue of *fairness* in schedules for systems of periodic tasks has received much attention. Recent research [4] has revealed the rather surprising fact that fairness can be achieved for the general PSP essentially “for free,” in the sense that (i) the feasibility conditions for ‘regular’ and fair scheduling are identical, and (ii) the runtime complexities of the corresponding scheduling algorithms are the same.

For various reasons, real-time systems practitioners often prefer static-priority scheduling schemes to dynamic-priority ones. We have addressed here the issue of achieving fairness within this framework. Our findings allow us to conclude that fairness can, once again, be achieved “for free,” and indicate that fair feasibility in static-priority systems may actually dominate ‘regular’ feasibility in such systems (e.g., Example 1). It is noteworthy that the asymptotic density-bound of $\ln 2$ (Corollary 4) was obtained using techniques fundamentally different from the ones used in [9], and indeed, the $\ln 2$ term appears from entirely different sources in the two results. Taken in conjunction with the results in [4], we conjecture that this is not accidental, and that our notion of fairness — pfairness — is in fact a very natural one for the Periodic Scheduling Problem.

Future Work. Intuitively speaking, the reason why multiprocessor scheduling is so much more difficult than uniprocessor scheduling is that greedy strategies (such as the earliest deadline first and rate monotonic algorithms) no longer seem to work. This is because adopting a greedy strategy may lead one into a situation where there are more processors available than active tasks, with the result that some processors are forced to be idle. As Liu pointed out, “the simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty” to the multiprocessor scheduling problem [8]. However, this scenario would

not occur if every task in the system had an execution requirement of one — for example, both the earliest deadline first and the rate monotonic algorithms can be easily modified to optimally schedule instances of the multiprocessor PSP where $x.e = 1$ for each task x . While pfair scheduling is not quite equivalent to the PSP with $x.e = 1$ for each task x , there are remarkable similarities. We intend to explore the issue of exploiting these similarities to design an optimal static pfair scheduling algorithm for the multiprocessor PSP. Since any pfair schedule is also a periodic schedule, we would then have a static-priority scheduling algorithm for solving the multiprocessor periodic scheduling problem, to complement the dynamic-priority algorithms presented in [4, 5].

References

- [1] M. Ajtai, J. Aspnes, M. Naor, Y. Rabani, L. Schulman, and O. Waarts. Fairness in scheduling. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1995.
- [2] Y. Azar, A. Broder, A. Karlin, and E. Ufal. Balanced allocations. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, May 1994.
- [3] A. Bar-Noy, A. Mayer, B. Schieber, and Madhu Sudan. Guaranteeing fair service to persistent dependent tasks. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1995.
- [4] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Proceedings of the ACM Annual Symposium on the Theory of Computing*, 1993. Accepted for publication in *Algorithmica*.
- [5] S. Baruah, J. Gehrke, and G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995. Extended version available via anonymous ftp from `ftp.cs.utexas.edu`, as Tech Report TR-95-02.
- [6] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.
- [7] L. Ford and D. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.
- [8] C. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary 37-60*, II:28–37, 1969.
- [9] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.
- [10] A. Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *Proc. 5th IEEE Workshop on Real-Time Software and Operating Systems*, Washington D.C., May 1988.
- [11] A. Mok and S. Ward. Distributed broadcast channel access. *Computer Networks*, 3:327–335, 1979.
- [12] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

Appendix

Proof of Theorem 3

The following two lemmas are rather technical in nature, and serve to set in place the machinery to prove Lemma 7. Their proofs may be skipped at a first reading.

Lemma 5 *An interval of size k contains no more than $\lceil k \cdot y.w \rceil$ activation-slots for task y , ($k \in \mathbf{N}$).*

Proof: Let $\text{earliest}(y, j+r) - \text{earliest}(y, j) + 1 = k$; i.e., the interval $[\text{earliest}(y, j), \text{earliest}(y, j+r)]$ is of size k . This interval contains $r+1$ activation-slots for task y . We will prove that $r+1 \leq \lceil k \cdot y.w \rceil$.

Let us suppose that the j 'th allocation to task y is done at slot $\text{earliest}(y, j) - 1$, in violation of the lag constraints. The lag at this slot is consequently some $\ell \leq -1$. Over the interval, the lag increases by $y.w$ at each slot, and decreases by 1 upon each allocation. The last slot in the interval is an activation-slot for task y ; allocating for the $(j+r)$ 'th time at this slot would consequently not drive the lag to below -1. That is,

$$\begin{aligned}
 \ell + k \cdot y.w - r &> -1 \\
 \Rightarrow -1 + k \cdot y.w - r &> -1 \quad /*\text{since } \ell \leq -1*/ \\
 \equiv r &< k \cdot y.w \\
 \equiv r + 1 &< k \cdot y.w + 1 \\
 \equiv r + 1 &\leq \lceil k \cdot y.w \rceil \\
 & \quad /*\text{since } r + 1 \text{ is an integer } */
 \end{aligned}$$

□

Observe that the bound of Lemma 5 is tight — i.e., an interval of size k may contain $\lceil k \cdot y.w \rceil$ activation-slots for task y . (Consider, for example, a task y with $y.w = 3/11$, and the interval $[3, 7]$ of size 5. Task y has $\lceil 5 \cdot \frac{3}{11} \rceil = 2$ activation-slots in this interval — at times 3 and 7.)

Lemma 6

$(\forall i, j : i \leq j : .$

$$(\text{latest}(x, j) - \text{earliest}(x, i) + 1) \geq \lfloor \frac{j - i + 1}{x.w} \rfloor .$$

Proof:

$$\begin{aligned} & \text{latest}(x, j) - \text{earliest}(x, i) + 1 \\ &= \lceil \frac{j+1}{x.w} \rceil - 1 - \lfloor \frac{i}{x.w} \rfloor + 1 \\ &\geq \frac{j+1}{x.w} - \frac{i}{x.w} \\ &= \frac{j-i+1}{x.w} \\ &\geq \lfloor \frac{j-i+1}{x.w} \rfloor \end{aligned}$$

□

Let $\chi(x) \stackrel{\text{def}}{=} \text{the maximum number of activation-slots by tasks with priority greater than } x\text{'s in any interval of size } \lfloor \frac{1}{x.w} \rfloor .$

Lemma 7 *For task x to not violate pfairness in the WM-generated schedule, it is sufficient that*

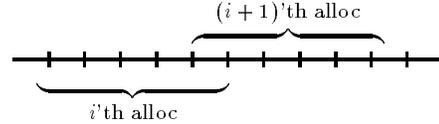
$$\chi(x) \leq \lfloor \frac{1}{x.w} \rfloor - 1. \tag{1}$$

Proof Sketch: In the spirit of [9], we define the *response time* of the i 'th request by task x to be the time span between the i 'th activation-slot for task x and the i 'th allocation to task x , $i \in \mathbf{N}$, and a *critical instant* for task x to be an instant at which a request for task x will have the largest response time. Using arguments similar to those used in [9, Theorem 1], it is not hard to see that this largest response time for task x is bounded by x 's response-time if all higher-priority tasks have activation-slots at the same slot as task x .

Suppose now that Condition 1 holds. By setting i equal to j in Lemma 6, we obtain the inequality $\text{latest}(x, i) - \text{earliest}(x, i) + 1 \geq \lfloor \frac{1}{x.w} \rfloor$. One might at first conclude that Lemma 7 immediately follows, since Condition 1 ensures that there are no more than $(\lfloor \frac{1}{x.w} \rfloor - 1)$ activation slots by higher-priority tasks

in the interval between a critical instant and the corresponding pseudo-deadline, thus leaving at least one slot for task x in this interval.

However, it has been proven [4] that $\text{latest}(x, i)$ — the i 'th pseudo-deadline for task x — may equal $\text{earliest}(x, i + 1)$ — the $(i + 1)$ 'th activation-slot for x . In other words, the intervals during which task x must be scheduled for the i 'th and the $(i + 1)$ 'th times may overlap by one slot:



To complete the proof of Lemma 7, therefore, we must consider the possibility that the slot not allocated to higher-priority tasks between the i 'th activation-slot and pseudo-deadline of x , and the one between the $(i + 1)$ 'th activation-slot and pseudo-deadline, are in fact one and the same. Fortunately, this possibility is ruled out by the following argument.

Consider any $j_o \in \mathbf{N}$. Let i_o be the largest i , $i \leq j_o$, such that $\text{latest}(x, i_o - 1) < \text{earliest}(x, i_o)$; if no such i exists, set i_o to zero. We will prove that the WM-generated schedule does not violate pfairness for task x over the interval $[\text{earliest}(x, i_o), \text{latest}(x, j_o)]$.

Let k range over $0, 1, \dots, j_o - i_o$. Observe that

$$\lfloor \frac{k+1}{x.w} \rfloor \geq (k+1) \lfloor \frac{1}{x.w} \rfloor \tag{2}$$

Suppose that allocations to task x corresponding to the activation-slots at $\text{earliest}(x, k')$ have been satisfied, for all $k' \in \{i_o, i_o + 1, \dots, i_o + k - 1\}$.

From Lemma 6, Equation 2 and Condition 1, we conclude that all the higher-priority tasks together leave at least $k + 1$ slots for task x over the interval $[\text{earliest}(x, i_o), \text{latest}(x, i_o + k)]$.

Furthermore, from Lemma 5, Lemma 6 (with $i \leftarrow i_o + k$, $j \leftarrow i_o + k$) and Equation 1, we conclude that all the higher-priority tasks together leave at least 1 slot for task x over the interval $[\text{earliest}(x, i_o + k), \text{latest}(x, i_o + k)]$.

It therefore follows that the allocation corresponding to the activation-slot at $\text{earliest}(x, i_o + k)$ is satisfied at or before $\text{latest}(x, i_o + k)$. □

We now determine sufficient conditions on the density of taskset Φ such that Condition 1 be satisfied. More specifically, we are looking for a bound on ρ — the density of Φ — such that any task system with a density no larger than this bound is guaranteed to satisfy Condition 1, and hence, is guaranteed to be

successfully scheduled by Algorithm WM. We start out by considering the case when Φ has only a few tasks.

Consider an instance Φ with two tasks x and y , with $y.w \geq x.w$, such that Algorithm WM assigns a greater priority to task y . Every request by task y will meet its pseudo-deadline.

Let $\lfloor 1/x.w \rfloor = k$; i.e., $\frac{1}{k+1} < x.w \leq \frac{1}{k}$. By Lemma 5 and the definition of $\chi(x)$, we may conclude that $\chi(x) \leq \lfloor k \cdot y.w \rfloor$. For Condition 1 to be violated, therefore, it is necessary that

$$\begin{aligned} \lfloor k \cdot y.w \rfloor &\geq k \\ \Rightarrow k \cdot y.w &> k - 1 \\ \equiv y.w &> 1 - \frac{1}{k}. \end{aligned}$$

The density of this infeasible instance is given by

$$\begin{aligned} \rho &= x.w + y.w \\ &> \frac{1}{k+1} + 1 - \frac{1}{k} \end{aligned}$$

which is minimized at $k = 2$, where it takes on value $1/2 + 1/3 = 5/6$ (since $x.w + y.w \leq 1$, and $x.w \leq y.w$, the possibility that $k = 1$ is ruled out).

The case when Φ contains three tasks is more interesting, and provides useful insight into deriving a density-bound for the general case. Since we expect the density-bound to be lower than for two tasks, assume that the 2 higher-density tasks — y_1 and y_2 — meet all their pseudo-deadlines, and consider the lowest-priority task x . Observe that $x.w \leq 1/3$ (in general, for Φ containing n tasks, the lowest-priority task x has density no larger than $1/n$).

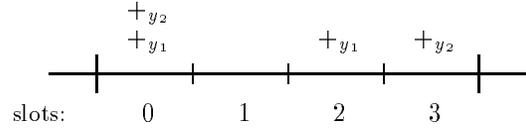
Case: ($1/4 < x.w \leq 1/3$). In this case, $\lfloor 1/x.w \rfloor = 3$. For Condition 1 to be violated, therefore, tasks y_1 and y_2 must together consume all 3 slots in an interval of size 3. Assuming $y_1.w \geq y_2.w$, we obtain, from Lemma 5,

$$\begin{aligned} \lfloor 3y_1.w \rfloor &\geq 2 \\ \Rightarrow y_1.w &> 1/3 \end{aligned}$$

Since $y_2.w \geq x.w$, it follows that the density of this infeasible instance is given by

$$\begin{aligned} \rho &= y_1.w + y_2.w + x.w \\ &> \frac{1}{3} + \frac{1}{4} + \frac{1}{4} \end{aligned} \quad (3)$$

Case: ($1/5 < x.w \leq 1/4$). In this case, $\lfloor 1/x.w \rfloor = 4$. For Condition 1 to be violated, tasks y_1 and y_2 must together consume all 4 slots in an interval of size 4. Suppose that they each consume 2 slots. Let $y_1.w \geq y_2.w$. Observe that task y_1 must have two activation-slots in an interval of size three (rather than four), since, if y_1 's and y_2 's second activation-slots were both in the fourth slot in the interval, then task x would be scheduled during the third slot:



(“+ y ” indicates that that slot is an activation-slot for task y .)

It therefore follows that

$$\begin{aligned} \lfloor 3y_1.w \rfloor &\geq 2 \Rightarrow y_1.w > 1/3 \\ \lfloor 4y_2.w \rfloor &\geq 2 \Rightarrow y_2.w > 1/4 \end{aligned}$$

The density of this infeasible instance is therefore given by

$$\begin{aligned} \rho &= y_1.w + y_2.w + x.w \\ &> \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \end{aligned} \quad (4)$$

which is smaller than the bound obtained above (Inequality 3).

It may be verified that choosing a less “even” distribution of activation slots for tasks y_1 and y_2 within the interval (e.g., y_1 gets three activation slots and y_2 just one), or a larger value of $\lfloor 1/x.w \rfloor$, does not yield a bound tighter than the one in Inequality 4. The density-bound for three-task systems is therefore $1/3 + 1/4 + 1/5$.

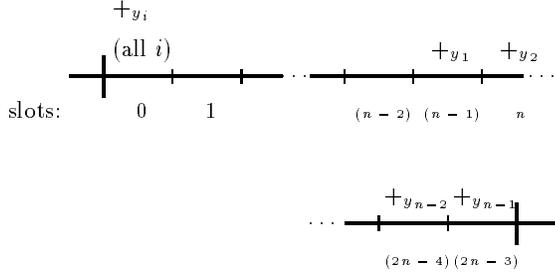
We are now ready to sketch out a proof for Theorem 3. Recall that Theorem 3 asserts:

Let Φ contain n tasks. If

$$\sum_{x \in \Phi} x.w \leq \sum_{i=n}^{2n-1} \frac{1}{i}$$

then Φ can be scheduled in a pfair manner by Algorithm WM.

Proof Sketch: Assume that the $n - 1$ higher-priority tasks in Φ all meet their pseudo-deadlines, and consider the lowest-priority task. Let task x be this lowest-priority task, and let tasks y_1, y_2, \dots, y_{n-1} be the higher-priority tasks, $y_i.w \geq y_{i+1}.w$, $1 \leq i <$



(“+ y ” on a slot indicates that that slot is an activation-slot for task y .)

Figure 2: Each higher-priority task consumes two slots within an interval of size $2n - 2$.

$(n - 1)$. For Condition 1 to be violated for task x , tasks y_1, y_2, \dots, y_{n-1} must together consume all the slots in an interval of size $\lceil 1/x.w \rceil$. A key observation is that the density-bound is lowest when each higher-priority task has an equal number of activation-slots in the interval. (Intuitively, a task that is to have a disproportionately high share of activation-slots in the interval will have its weight increase by more than the corresponding decrease in the weight of a task having fewer activation slots, when compared to an instance in which each task has an equal number of slots.) Since we must have $\lceil 1/x.w \rceil \geq n$, and there are only $n - 1$ higher-priority tasks, the possibility of an infeasible instance, in which each higher-priority task has just one activation-slot within an interval of size $\lceil 1/x.w \rceil$, is ruled out. The bound in Theorem 3 is obtained by considering two activation-slots per higher-priority task in an interval of size $\lceil 1/x.w \rceil$ (Figure 2). For this to occur, we require that $\lceil 1/x.w \rceil = 2(n - 1)$; i.e., $1/(2n - 2) \geq x.w > 1/(2n - 1)$. It may be verified that having more than two activation slots per higher-priority task yields a larger bound than the one in Theorem 3.

Suppose that each task y_i consumes two slots (Figure 2). It follows that

$$\begin{aligned}
 \lceil n \cdot y_1.w \rceil \geq 2 &\Rightarrow y_1.w > 1/n \\
 \lceil (n + 1) \cdot y_2.w \rceil \geq 2 &\Rightarrow y_2.w > 1/(n + 1) \\
 \lceil (n + 2) \cdot y_3.w \rceil \geq 2 &\Rightarrow y_3.w > 1/(n + 2) \\
 &\dots \\
 \lceil (n + i - 1) \cdot y_i.w \rceil \geq 2 &\Rightarrow y_i.w > 1/(n + i - 1) \\
 &\dots
 \end{aligned}$$

The density of this infeasible instance is therefore given by

$$\begin{aligned}
 \rho &= \sum_{i=1}^{n-1} y_i.w + x.w \\
 &> \sum_{i=n}^{2n-2} \frac{1}{i} + \frac{1}{2n-1} \\
 &= \sum_{i=n}^{2n-1} \frac{1}{i}
 \end{aligned}$$

□