

Pfair Scheduling of Sporadic Tasks*

James H. Anderson and Anand Srinivasan

Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175

Abstract

Almost all prior work on Pfair scheduling has been limited to synchronous, periodic task systems. In this paper, we sketch a new proof that shows that the PD Pfair algorithm is optimal for scheduling sporadic tasks on a multiprocessor.

1 Introduction

Pfair scheduling was proposed by Baruah et al. as a way of optimally and efficiently scheduling periodic tasks on a multiprocessor system [2, 3]. Pfair scheduling differs from more conventional real-time scheduling disciplines in that tasks are explicitly required to make progress at steady rates. In the classic periodic task model, each task T executes at an implicit rate given by $T.e/T.p$, where $T.e$ is the *execution cost* of each job of T , and $T.p$ is the *period* of T . However, this notion of a rate is a bit inexact: a job of T may be allocated $T.e$ time units at the beginning of its period, or at the end of its period, or its computation may be spread out more evenly. Under Pfair scheduling, this implicit notion of a rate is strengthened to require each task to be executed at a rate that is uniform across each job.

Pfair scheduling algorithms ensure uniform execution rates by breaking tasks into quantum-length “subtasks.” Each subtask must execute within a “window” of time slots, the last of which is its deadline. These windows divide each period of a task into subintervals of approximately equal length. By breaking tasks into smaller executable units, Pfair scheduling algorithms circumvent many of the bin-packing-like problems that lie at the heart of intractability results involving multiple-resource real-time scheduling problems. Intuitively, it is easier to evenly distribute small, uniform items among the available bins than larger, non-uniform items.

One limitation of most prior work on Pfair scheduling is that only synchronous, periodic task systems have been considered. This stands in contrast to most

uniprocessor scheduling schemes, where demand-based arguments can be applied to easily extend correctness proofs for periodic task systems to also apply to sporadic task systems. As explained later, such demand-based arguments do not work in Pfair-scheduled systems. Instead, Pfair algorithms are usually proved correct by means of arguments that hinge on the exact alignment of future windows. Unfortunately, such alignments cannot always be predicted in sporadic task systems, because some jobs may be released “late.” This leaves us in a quandary: Simple demand-based reductions will not work, yet the techniques used in most previous proofs for Pfair algorithms are not easy to extend to the sporadic case.

In this paper, we sketch a new proof that shows that the PD Pfair algorithm is optimal for scheduling sporadic tasks on a multiprocessor. PD is the most efficient Pfair algorithm proposed to date (for synchronous, periodic systems). The proof consists of two steps. First, we show that PD can miss a deadline only if it previously left a processor idle where an optimal scheduler would not. Second, we use a swapping argument to show that PD cannot leave more processors idle than an optimal scheduler, giving a contradiction. This swapping argument involves a prefix of the schedule in which window alignments *can* be sufficiently predicted.

The rest of this paper is organized as follows. In Section 2, we define Pfair scheduling and briefly describe the PD algorithm. In Section 3, we sketch the proof mentioned above.

2 Pfair Scheduling

Consider a collection of synchronous, periodic real-time tasks to be executed on a system of multiple processors.¹ We assume that processor time in such a system is allocated in discrete time units, or quanta; the time interval $[t, t + 1)$, where t is a nonnegative integer, is called *slot* t . Associated with each task T is a *period* $T.p$ and an *execution cost* $T.e$. Every $T.p$

*Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, and ITR 0082866.

¹The term *synchronous* means that each task begins execution at time 0. Sporadic tasks will be considered later.

time units, a new invocation of T with a cost of $T.e$ time units is released into the system; we call such an invocation a *job* of T . Each job of a task must complete execution before the next job of that task begins. Thus, $T.e$ time units must be allocated to T in each interval $[k \cdot T.p, (k + 1) \cdot T.p)$, where $k \geq 0$. T may be allocated time on different processors in such an interval, as long as it is not allocated time on different processors at the same time.

The sequence of allocation decisions over time defines a “schedule.” Formally, a *schedule* S is a mapping $S : \tau \times \mathcal{N} \mapsto \{0, 1\}$, where τ is a set of periodic tasks and \mathcal{N} is the set of natural numbers. If $S(T, t) = 1$, then we say that T is *scheduled at slot* t .

Lag constraints. The ratio $T.e/T.p$ is called the *weight* of task T , denoted $wt(T)$. We assume each task’s weight is strictly less than one — a task with weight one would require a dedicated processor, and thus is quite easily scheduled. A task with weight less than $1/2$ is called a *light* task, while a task with weight at least $1/2$ is called a *heavy* task.

A task’s weight defines the rate at which it is to be scheduled. Because processor time is allocated in quanta, we cannot guarantee that a task T will execute for *exactly* $(T.e/T.p)t$ time during each interval of length t . Instead, in a Pfair-scheduled system, processor time is allocated to each task T in a manner that ensures that its rate of execution never deviates too much from $T.e/T.p$. More precisely, correctness is defined by focusing on the *lag* between the amount of time allocated to each task and the amount of time that would be allocated to that task in an ideal system with a quantum approaching zero. The *lag of task* T at time t , denoted $lag(T, t)$, is defined as follows:

$$lag(T, t) = (T.e/T.p)t - allocated(T, t), \quad (1)$$

where $allocated(T, t)$ is the amount of processor time allocated to T in $[0, t)$. A schedule is *Pfair* iff

$$(\forall T, t :: -1 < lag(T, t) < 1). \quad (2)$$

Informally, the allocation error associated with each task must always be less than one quantum. It is straightforward to show that any Pfair schedule is also periodic [1, 3]. Baruah et al. [3] showed that a periodic task set τ has a Pfair schedule on M processors iff

$$\sum_{T \in \tau} \frac{T.e}{T.p} \leq M. \quad (3)$$

Windows. The Pfair lag bounds given in (2) have the effect of breaking each task T into an infinite se-

quence of unit-time *subtasks*. We denote the i^{th} subtask of task T as T_i , where $i \geq 1$. As in [2], we associate with each subtask T_i a *pseudo-release* $r(T_i)$ and a *pseudo-deadline* $d(T_i)$. If T_i is synchronous and periodic, then $r(T_i)$ and $d(T_i)$ are as follows.

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \quad (4)$$

$$d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil - 1 \quad (5)$$

(Derivations of these expressions can be found in [1].) $r(T_i)$ is the first slot into which T_i potentially could be scheduled, and $d(T_i)$ is the last such slot. For brevity, we often refer to pseudo-deadlines and pseudo-releases as simply deadlines and releases, respectively. The interval $[r(T_i), d(T_i)]$ is called the *window* of subtask T_i and is denoted by $w(T_i)$. The *length* of window $w(T_i)$, denoted $|w(T_i)|$, is defined as $d(T_i) - r(T_i) + 1$.

As an example, consider a task T with weight $wt(T) = 8/11$. Each job of this task consists of eight windows, one for each of its unit-length subtasks. Using Equations (4) and (5), it is possible to show that the windows within each job of T are as depicted in Figure 1. Note that successive windows of T overlap only by one slot. In general, consecutive windows of a task are either disjoint or overlap by one slot [1].

Sporadic task systems. In a sporadic tasks system, the first job of a task may be released after time 0, and there may be separation between consecutive jobs of the same task. Thus, for each job J of a task T , an *offset* can be defined, which is the difference between J ’s release time in the sporadic system and J ’s release time in a corresponding synchronous, periodic system. For example, if T has a period of 10, and its first two jobs are released at times 5 and 20, respectively, then its first job has an offset of 5, and its second job has an offset of 10. Such offsets can be used to adjust the formulae above for sporadic systems. For instance, (4) becomes

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor + \Delta(T_i),$$

where $\Delta(T_i)$ is the offset of the job containing T_i . In the definition of lag in Equation (1), offsets would be subtracted from the right-hand side to ensure that T has zero lag when it has no enabled job. The feasibility condition (3) also can be shown to hold for sporadic systems.

Scheduling algorithms. For synchronous, periodic task systems, the most efficient Pfair scheduling al-

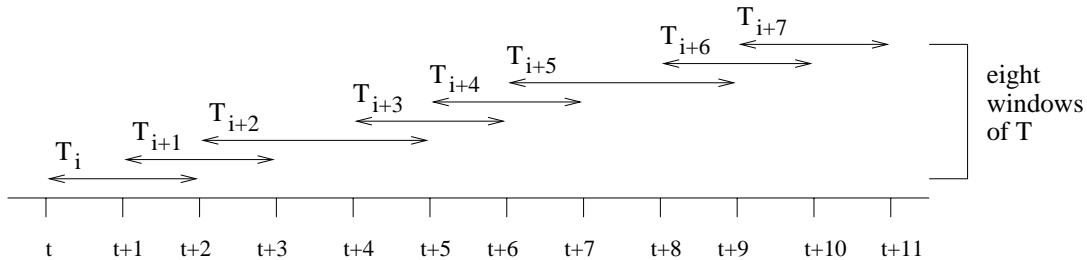


Figure 1: The eight “windows” of a task T with weight $wt(T) = 8/11$. Under Pfair scheduling, each of T 's eight units of computation must be allocated processor time during its window, or a lag-bound violation will result.

gorithm proposed to date is an algorithm called PD [2, 3]. PD schedules subtasks by pseudo-deadline. In the original PD algorithm, four tie-break parameters were used to resolve ties among subtasks with the same deadline. In recent work, we proved that only two tie-break parameters suffice [1]. Due to space limitations, these tie-break parameters are not described here; a detailed explanation of them can be found in [1].

3 Optimality of PD for Sporadic Systems

Because synchronous, periodic task systems represent a “worst-case” scenario in the spectrum of sporadic task systems, one may think that the optimality of PD for sporadic tasks would follow as a simple corollary from previous work. However, the correctness proofs given for PD in [1] and [3] rely *crucially* on the fact that at any moment of time, future window alignments can be predicted. As mentioned previously, such predictions cannot be made for sporadic task systems.

Why demand-based arguments don't work.

For uniprocessor scheduling algorithms, one can often reduce sporadic systems to asynchronous, periodic systems in the following way. Consider a scheduling algorithm A that has been shown to be correct for asynchronous, periodic tasks. Suppose that there exists a feasible sporadic task system τ that misses a deadline at some time t_d when scheduled using A . Let S be the corresponding schedule. We may assume that all jobs in S after t_d are released in a periodic fashion, because such jobs have no impact on the deadline miss at time t_d . Now, if we inductively “right-shift” all jobs released before time t_d in S until there are no sporadic separations among jobs before t_d , then we get a schedule S' that is in accordance with the asynchronous, periodic task model (see Figure 2(a)). Moreover, “right-shifting” such jobs in S can only increase demand near

time t_d . Thus, a deadline must be missed at time t_d , a contradiction.

To see why this argument cannot be applied in a Pfair-scheduled system, consider Figure 2(b). Here, subtask T_i is right-shifted into slot t . Before the shift, subtask U_j was scheduled at t and some processor was idle at t . After the shift, U_j has higher priority than T_i , so the two are swapped in the schedule as a result of shifting T_i . Note that U_j being scheduled at t makes U_{j+1} ineligible at t . However, after T_i and U_j are swapped, U_{j+1} is eligible at t and thus it may left-shift into slot t . This may cause a cascade of other left-shifts, and hence a presumed future deadline miss may be eliminated. The root of the problem here is that right-shifting jobs (and hence subtasks) may in fact *decrease* future demand.

Our approach. Because of such difficulties, our proof is based upon a different approach. Suppose that there exists a sporadic task set τ that misses a deadline at time t_d under PD, yet there exists a valid schedule for τ such that no deadline is missed at t_d . For this to happen, PD must have left more processors idle before t_d than in the valid schedule. We use the term “hole” to refer to idle processors in the schedule. Specifically, in a schedule S , if k processors are idle at time t , then we say that there are k holes in S at time t . The key observation upon which our proof is based is this: If PD is incorrect, then its failure is really due to the fact that it creates holes where an optimal scheduler would not. The fact that a deadline is subsequently missed is just an unhappy consequence of this.

In particular, over the interval $[0, t_d]$, PD schedules at least one subtask less than an optimal scheduler would, namely the one that misses its deadline. This implies the existence of a time $t_h < t_d$ such that the number of holes over $[0, t_h]$ in the optimal schedule is strictly less than the number of holes over $[0, t_h]$ in the PD schedule. The value of t_h is strictly smaller

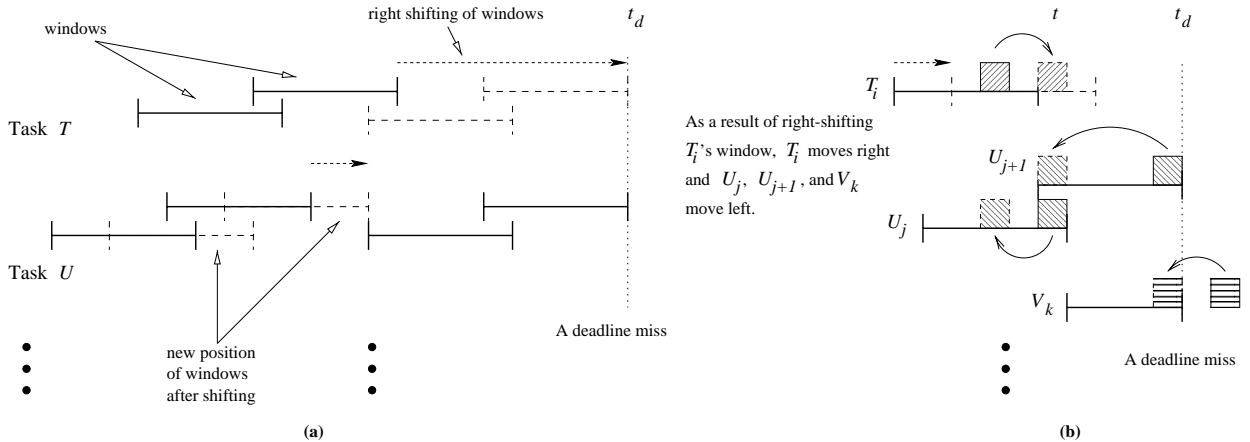


Figure 2: (a) Starting with a sporadic task set that misses a deadline at t_d , we can right-shift all windows towards t_d . Intuitively, we *should* get an asynchronous, periodic task system that misses a deadline at t_d . (b) Unfortunately, right-shifting a window need not increase future demand. Here, shifting T_i to the right by two slots decreases its priority and hence subtask U_j , which was scheduled later at time t in the original schedule, may now have higher priority. Note that at time t , a processor might be idle, in which case U_{j+1} can now be scheduled in that slot. This can cause a cascade of future left-shifts. Thus, right-shifting a window can in fact *decrease* future demand.

than t_d , because there cannot be a hole at t_d in the PD schedule. (This would contradict the fact that PD misses a deadline at time t_d .) This reasoning is encapsulated in the following lemma.

Lemma 1 *Let τ be a feasible sporadic task system that misses a deadline at time t_d under PD. Let S_{PD} denote the corresponding schedule. Let S_{OPT} be a schedule for τ generated by an optimal scheduler. Then, there exists a time t_h , where $t_h < t_d$, such that the number of holes over $[0, t_h]$ in S_{PD} is more than the number of holes over $[0, t_h]$ in S_{OPT} .*

Because PD is a “greedy” scheduler, we can show that it will not create more holes than the optimal scheduler, thus giving us a contradiction. In the full proof, this is shown by means of a swapping argument in which, starting with a valid schedule generated by an optimal scheduler, subtasks are inductively swapped until the schedule is in accordance with PD. Each swapping satisfies the following properties.

- The resulting schedule is valid.
- The swapping results in one more subtask being in accordance with the PD priority definition.
- The number of holes in each slot remains unchanged.

From these properties, we conclude that PD cannot create “extra” holes. Thus, we have the following.

Lemma 2 *For any sporadic task system τ , the number of holes produced by PD over $[0, t_h]$ is at most the number of holes produced by an optimal scheduler over $[0, t_h]$.*

The proof of this lemma is actually quite long (about eight single-spaced pages). This is because a considerable number of cases must be considered to handle all possible swappings. These cases depend on future window alignments, whether the tasks involved in the swapping are heavy or light, etc.

Combining Lemmas 1 and 2, we have the following.

Theorem 1 *PD correctly schedules any feasible sporadic task system.*

References

- [1] J. Anderson and A. Srinivasan. A new look at pfair priorities. Technical Report TR00-023, University of North Carolina at Chapel Hill, Sept. 2000. Available at <http://www.cs.unc.edu/~anderson/papers.html>.
- [2] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.
- [3] S. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proc. of the 9th International Parallel Processing Symposium*, pages 280–288, Apr. 1995.