Integrating Aperiodic and Recurrent Tasks on Fair-scheduled Multiprocessors[*]

Anand Srinivasan, Phil Holman, and James H. Anderson

Department of Computer Science, University of North Carolina

Chapel Hill, NC 27599-3175

E-mail: {anands,holman,anderson}@cs.unc.edu

December 2001

Abstract

We propose two server implementations for multiplexing aperiodic and recurrent (*i.e.*, periodic, sporadic, or "rate-based") real-time tasks in fair-scheduled multiprocessor systems. This is the first paper to consider the problem of integrating support for aperiodic tasks within fair multiprocessor scheduling algorithms. We also provide admission-control tests for the scheduling of *hard* aperiodic tasks (which have deadlines). Further, we point out some of the additional complexities involved in server-based implementations on multiprocessors and present some ways to handle them. Most of these complexities arise because of the parallelism that exists in such systems. Finally, we provide experimental results that demonstrate the effectiveness of our implementations.

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

1 Introduction

There has been much recent work on scheduling techniques for multiplexing aperiodic and recurrent tasks in uniprocessor systems [8, 9, 11, 13, 14, 15, 16]. A *recurrent task* releases successive *jobs* (according to some specified rules) that are subject to deadlines. On the other hand, an *aperiodic task* is a "one-shot" task that arrives at an arbitrary time. Such a task may be either *hard* or *soft* — a hard aperiodic task has a deadline, while a soft aperiodic task does not. Whether an arriving hard aperiodic task's deadline can be guaranteed is a function of the current system workload; thus, such tasks must be subject to an admission-control test. Usually, the goal is to admit as many hard aperiodic tasks as possible and to minimize the response times of soft aperiodic tasks, without causing recurrent tasks to miss their deadlines.

All of the work cited above is directed at the problem of multiplexing aperiodic and recurrent tasks on a *single* resource and thus cannot be applied in multiplexed systems with workloads that exceed the capacity of a single processor. Such systems are becoming commonplace. Consider, for example, the proliferation of Internet service providers that host third-party websites on multiprocessor servers [6]; such websites may include sites that stream audio and video and hence require real-time guarantees.

In this paper, we present several server-based schemes for implementing multiprocessor systems such as these. These schemes build upon recent work on fair multiprocessor scheduling. In fair scheduling disciplines, each task is guaranteed to receive a specific share of the resource(s) to be scheduled. As explained below, this notion of fairness may or may not be work-conserving. We base our work on fair scheduling mainly because it is the only known optimal way for scheduling recurrent real-time tasks on multiprocessors. In addition, the associated feasibility condition is simple and computationally efficient. Finally, such algorithms are of practical importance. As a case in point, Ensim Corp., an Internet service provider, has deployed multiprocessor fair scheduling algorithms in its product line. These algorithms have been evaluated empirically by Chandra *et al.* [6, 7]. Although the results of Chandra *et al.* demonstrate convincingly the utility of using fair scheduling algorithms to multiplex multiple applications on multiprocessor servers, they did not explicitly address the problem of integrating aperiodic and recurrent tasks, nor did they formally analyze any of the algorithms they considered. In contrast, the results in this paper are based on algorithms that *have* been formally verified.

In the following paragraphs, we describe the fair scheduling disciplines of relevance to our work in greater detail. After that, we present a more detailed overview of the contributions of this paper.

Scheduling schemes considered in this paper. The simplest notion of a recurrent task is that provided by the *periodic* task model. In this model, successive job releases by the same task are spaced apart by a fixed interval, called the task's *period*. Periodic task systems can be optimally scheduled on multiprocessors using *Pfair* scheduling algorithms [3, 4, 5]. Under Pfair scheduling, each task is required to execute at a uniform rate, while respecting a fixed allocation quantum. Uniform rates are ensured by requiring the allocation error for each task to be always less than one quantum, where "error" is determined by comparing to an ideal fluid system. Due to this requirement, each task is effectively subdivided into quantum-length *subtasks* that must

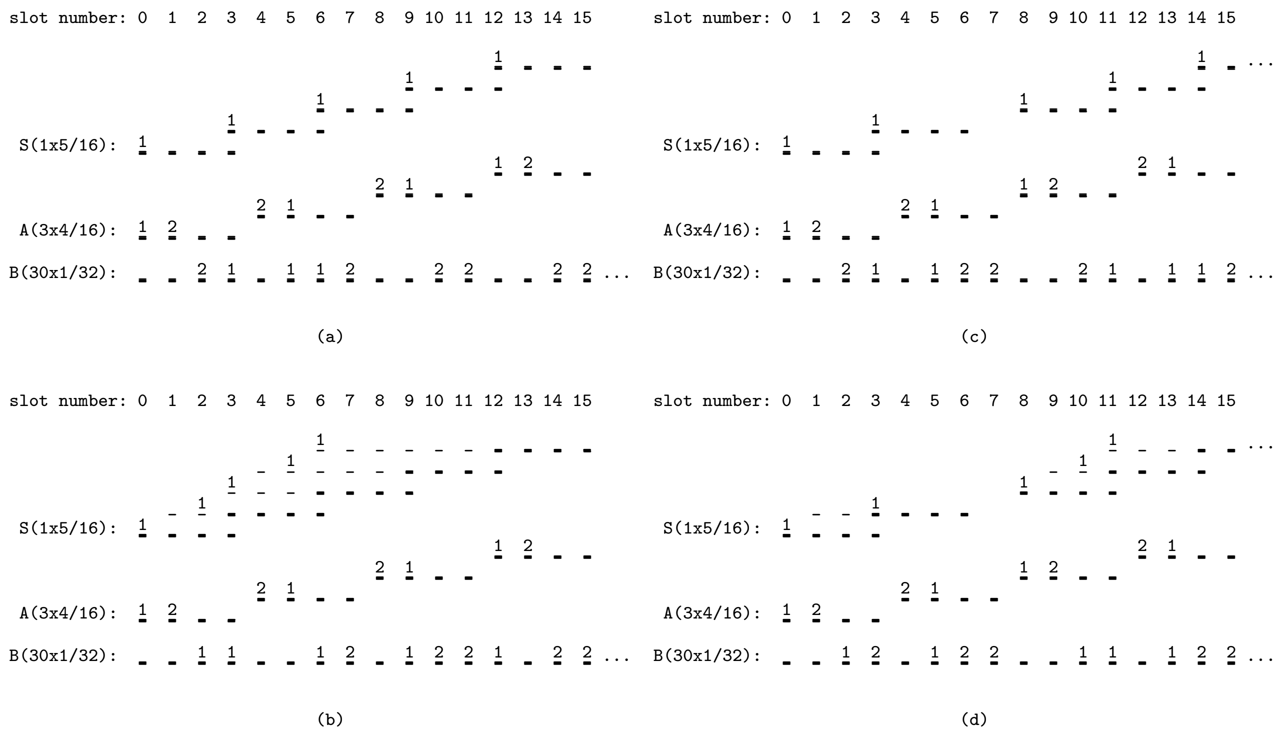


Figure 1: A partial schedule on two processors for a task set is shown under various conditions. In each schedule, tasks of a given weight are shown together. Each unit-time subtask has an *eligibility interval*, denoted by dashes, corresponding to the sequence of time slots (*i.e.*, quanta) in which it can be scheduled; a subtask's eligibility interval must include its Pfair window, denoted in bold. An integer value n in slot t of some window means that n of the subtasks that must execute within that window are scheduled in slot t. No integer value means that no such subtask is scheduled in slot t. **(a)** All tasks are periodic and are Pfair-scheduled. **(b)** All tasks are periodic, task S is ERfair-scheduled and all other tasks are Pfair-scheduled. **(c)** The third subtask of task S is released late; all tasks are Pfair-scheduled. **(d)** The third subtask of task S is released late, task S is ERfair-scheduled, and the others are Pfair-scheduled.

execute within *windows* of approximately equal lengths: if a subtask of a task T executes outside of its window, then T's error bounds are exceeded. A task's subtasks may execute on any processor, *i.e.*, migration is allowed.

As an example, consider the two-processor schedule in Fig. 1(a). (The other insets of this figure are considered below.) The length and alignment of a recurrent task's Pfair windows is determined by its *weight*, which is defined as the ratio of its per-job execution cost and period. The depicted schedule includes a task S of weight 5/16, a set A of three tasks of weight 4/16 each, and a set B of 30 tasks of weight 1/32 each.

Under Pfair scheduling, if some subtask of a task T executes "early" within its window, then T is ineligible for execution until the beginning of its next window. This means that Pfair scheduling algorithms are necessarily not work conserving when used to schedule periodic tasks. Informally, a scheduling algorithm is *work conserving* if no processor ever idles unnecessarily. In [1], Anderson and Srinivasan proposed a work-conserving variant of Pfair scheduling, called *Early-release fair* (*ERfair*) scheduling, and showed that ERfair scheduling algorithms also can be used to optimally schedule periodic tasks. ERfair scheduling differs from Pfair scheduling in a rather simple way: under ERfair scheduling, a subtask may become eligible for execution *early*, *i.e.*, before its Pfair window. By allowing early releases, response times can often be reduced. This is illustrated in Fig. 1(b); note that S completes six time units earlier here than an inset (a).

In recent work, Anderson and Srinivasan showed that sporadic tasks can also be scheduled optimally on multiprocessors using Pfair and ERfair scheduling algorithms [1, 2, 3, 17]. In the sporadic model, the notion of recurrence is relaxed by specifying a minimum (rather than exact) spacing between consecutive job releases of the same task. In [2], the sporadic model was extended to obtain the *intra-sporadic* model, which has many characteristics in common with the recently-proposed uniprocessor rate-based execution model [12]. In essence, the sporadic model allows *jobs* to be released "late"; the intra-sporadic model allows *subtasks* to be released "late," as illustrated in insets (c) and (d) of Fig. 1. In addition, early subtask-releases may be allowed, as in ERfair scheduling. As shown in [2, 17], Pfair and ERfair scheduling algorithms can be used to optimally schedule intra-sporadic tasks. Since the intra-sporadic model subsumes the other models discussed above, we take it as our notion of "recurrence" in this paper, unless otherwise specified. As we shall see later, the flexibility inherent in this notion of recurrence makes the intra-sporadic model ideal for defining aperiodic servers.

Contributions. We present two server-based approaches for scheduling aperiodic tasks on multiprocessors. We also present an admission-control test for hard aperiodic tasks; the analysis associated with this test can also be used to determine worst-case response times for soft aperiodic tasks. Further, we consider some of the difficulties that arise when scheduling aperiodic tasks on multiprocessors and present some ways to handle them. We also demonstrate the effectiveness of our server implementations by simulation experiments that compare them to background scheduling.

The server-based algorithms that we consider are two-level schedulers: one or more servers are scheduled along with the intra-sporadic ("recurrent") real-time tasks in the system, and these servers in turn schedule the aperiodic tasks — how many servers to use, their weights, and how to divide the aperiodic tasks among them are among the issues unique to multiprocessor systems that must be considered. The two server-based approaches we consider differ in that Pfair scheduling techniques are used in one and ERfair in the other. As we shall see, using ERfair-scheduled servers usually results in lower aperiodic response times. For each approach, we consider three variants. These variants differ in the server's behavior when no no aperiodic tasks are ready. Insets (c) and (d) of Fig. 1 illustrate the difference between using Pfair- and ERfair-scheduled servers. In both insets, task S represents an aperiodic server. The server services two aperiodic tasks — one with execution requirement 2 released at time 0, and another with execution requirement 3 released at time 8. Note that the second task completes three time slots earlier in the ERfair-server case.

In addition to these server schemes, we also consider the use of *background servers* on each processor. Such a server is scheduled when its processor becomes idle and aperiodic tasks are available to schedule. These servers are necessary in order to fully exploit the parallelism that exists in the system.

Organization. In Sec. 2, the scheduling disciplines we consider are defined formally. In Sec. 3, the two server schemes mentioned above are presented in detail, assuming there is a single server in the system. The multiple-server case is considered in Sec. 4. In Sec. 5, an empirical evaluation of the proposed schemes is presented. Sec. 6 concludes the paper.

2 Definitions

In the following subsections, relevant definitions are given. We begin with Pfair and ERfair scheduling.

2.1 Pfair and ERfair Scheduling

In defining notions relevant to Pfair and ERfair scheduling, we limit attention (for now) to periodic task systems. A periodic task T with an integer *period* $T.p$ and an integer *execution cost* $T.e$ has a *weight* of $T.e/T.p$, denoted $wt(T)$. Every $T.p$ time units, T releases a new *job* (*i.e.*, instance) with cost $T.e$. Each job of T must complete execution before the next job of T is released. We require that $T.e \leq T.p$ and hence $wt(T) \leq 1$. A task with weight less than (at least) 1/2 is called a *light* (*heavy*) task.

Pfair and ERfair scheduling algorithms allocate processor time in discrete quanta; the time interval $[t, t+1)$, where t is a nonnegative integer, is called *slot* t. (Hence, *time* t refers to the beginning of slot t.) A task may be allocated time on different processors, but not in the same slot. The notion of a Pfair schedule is defined by comparing to an ideal system that allocates $(T.e/T.p)t$ time to each task T over $[0, t]$. Deviance from the ideal system is formally captured by the concept of *lag*. Formally, the *lag of task T at time t in a schedule S* is $\mathit{lag}(T, t, S) = (T.e/T.p)t - \mathit{allocated}(T, t)$, where $\mathit{allocated}(T, t)$ is the amount of processor time allocated to T in $[0, t)$. A schedule S is *Pfair* iff

$$(\forall T, t :: -1 < \mathit{lag}(T, t, S) < 1). \tag{1}$$

Informally, the allocation error associated with each task must always be less than one quantum.

The lag bounds above have the effect of breaking each task T into an infinite sequence of unit-time *subtasks*. We denote the i^{th} subtask of task T as T_i, where $i \geq 1$. As in [4], we associate a *pseudo-release* $r(T_i)$ and *pseudo-deadline* $d(T_i)$ with each subtask T_i, as follows. (For brevity, we often drop the prefix "pseudo-.")

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{2}$$

$$d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil - 1 \tag{3}$$

(Note that these definitions refer to slots, which are numbered from 0.) In a Pfair-scheduled system, subtask T_i can be scheduled only within the interval $[r(T_i), d(T_i)]$, termed its *window*, and denoted $w(T_i)$. The *length* of $w(T_i)$, denoted $|w(T_i)|$, is $d(T_i) - r(T_i) + 1$. As an example, consider a task T with weight $T.e/T.p = 8/11$. Each job of T consists of eight windows, one for each of its unit-length subtasks, as shown in Fig. 2. It can be shown that, in general, consecutive windows of a task are either disjoint or overlap by one slot.

The notion of ERfair scheduling [1] is obtained by simply dropping the -1 constraint in (1). Formally, a schedule S is *ERfair* iff $(\forall T, t :: \mathit{lag}(T, t, S) < 1)$. With this change, a subtask can become eligible before its window, as illustrated earlier in Fig. 1(b). Note that any Pfair schedule is ERfair, but not necessarily vice versa. It is easy to show that, in any Pfair or ERfair schedule, all job deadlines are met [1].

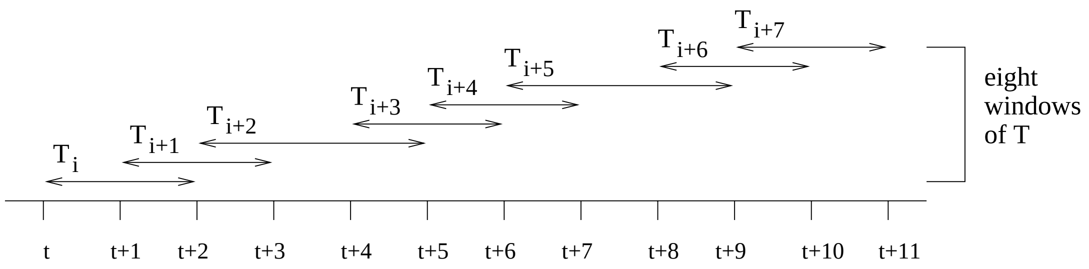


Figure 2: Pfair scheduling: the eight "windows" of a periodic task T with weight $T.e/T.p = 8/11$. Each of T's eight units of computation must be allocated processor time during its window, or else a lag-bound violation will result.

2.2 Intra-sporadic Tasks

The intra-sporadic task model generalizes the sporadic model by allowing separation between consecutive subtasks of a task. In addition, early releases are allowed. Formally, an *intra-sporadic task system* is defined by a pair (τ, e), where τ is a task set, and e is a function that indicates when each subtask becomes eligible. Each task may release either a finite or infinite number of subtasks. We assume that $e(T_i) \geq e(T_{i-1})$ for all $i > 1$.

In the intra-sporadic model, each subtask has both a *Pfair window* (PF-window) and an *intra-sporadic window* (IS-window). A subtask's IS-window, which includes its PF-window, defines the interval during which it is eligible to be scheduled. Formally, T_i's IS-window is defined as $[e(T_i), d(T_i)]$ and its PF-window is defined as $[r(T_i), d(T_i)]$. As we shall see, the terms $r(T_i)$ and $d(T_i)$ have a similar interpretation to that given previously for periodic task systems. As before, we will use $w(T_i)$ to denote the PF-window of subtask T_i.

$r(T_i)$ and $d(T_i)$ are defined inductively by examining the PF-windows of T in a *periodic* system. As mentioned earlier, consecutive PF-windows of a periodic task are either disjoint or overlap by one slot. The bit $b(T_i)$ distinguishes between these two possibilities.

$$b(T_i) = \begin{cases} 1, & \text{if } \left\lceil \frac{i}{wt(T)} \right\rceil = \left\lfloor \frac{i}{wt(T)} \right\rfloor + 1 \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

By (2) and (3), if T is *periodic*, then $w(T_i)$ and $w(T_{i+1})$ overlap (by one slot) if $b(T_i) = 1$, and do not overlap if $b(T_i) = 0$. For intra-sporadic tasks, we define $b(T_i)$ exactly as above. Given this definition, we can define $r(T_i)$, which defines the beginning of T_i's PF-window.

$$r(T_i) = \begin{cases} e(T_i), & \text{if } i = 1 \\ max(e(T_i), d(T_{i-1}) + 1 - b(T_{i-1})), & \text{if } i \geq 2 \end{cases} \tag{5}$$

Thus, if T_i becomes eligible *during* T_{i-1}'s PF-window, then $r(T_i) = d(T_{i-1}) + 1 - b(T_{i-1})$, and hence, the spacing between $r(T_{i-1})$ and $r(T_i)$ is exactly as in a periodic task system.[1] On the other hand, if T_i becomes

[1]Note that the notion of a job is not mentioned here. For systems in which subtasks are grouped into jobs that are released in sequence, the definition of e would preclude a subtask from becoming eligible before the beginning of its job.

eligible *after* T_{i-1}'s PF-window, then T_i's PF-window begins when T_i becomes eligible. Note that (5) implies that consecutive PF-windows of the same task are either disjoint or overlap by one slot, as in a periodic system.

T_i's deadline $d(T_i)$ is defined as $r(T_i) + |w(T_i)| - 1$, where $|w(T_i)|$ is the length of its PF-window in a periodic system. Thus, by (2) and (3),

$$|w(T_i)| = \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{6}$$

$$d(T_i) = r(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor - 1. \tag{7}$$

Hence, if T_i becomes eligible early, *i.e.*, $e(T_i) < r(T_i)$, then its deadline is postponed to where it would have been if $e(T_i) = r(T_i)$.

To summarize, our notion of an intra-sporadic task is obtained by allowing a task's PF-windows to be right-shifted from where they would appear if the task were periodic; thus, another way to calculate the deadline of a subtask is to use Equation (3) and add an *offset* that gives the amount by which the PF-window has been right-shifted. In addition, we allow a subtask to become eligible before its PF-window, as in ERfair scheduling. A schedule for a system of intra-sporadic tasks is said to be *valid* iff each subtask is scheduled in its IS-window.

As shown in [2], an intra-sporadic task system τ has a valid schedule on M processors iff

$$\sum_{T \in \tau} \frac{T.e}{T.p} \leq M. \tag{8}$$

It is easy to see that the intra-sporadic model generalizes the sporadic model; hence, the above inequality is a feasibility condition for sporadic tasks as well.

2.3 The PD^2 Algorithm

The PD^2 algorithm is the most efficient Pfair or ERfair scheduling algorithm proposed to date, and is optimal for all the task models considered in this paper [1, 2, 3, 17]. PD^2 prioritizes subtasks by their deadlines. Any ties are broken using two tie-break parameters: the b-bit defined in (4), and the "group deadline," defined next.

The group deadline. It can be shown that all windows of a heavy task are of length two or three. Consider a sequence $T_i, \ldots, T_j$ of subtasks of a heavy task T (without any late releases) such $|w(T_k)| = 2$ for all $i < k \leq j$, $b(T_k) = 1$ for all $i \leq k < j$, and either $b(T_j) = 0$ or $|w(T_{j+1})| = 3$ (*e.g.*, T_i, T_{i+1} or T_{i+2}, T_{i+3}, T_{i+4} or T_{i+5}, T_{i+6}, T_{i+7} in Fig. 2). If any of $T_i, \ldots, T_j$ is scheduled in the last slot of its window, then each subsequent subtask in this sequence must be scheduled in its last slot. In effect, $T_i, \ldots, T_j$ must be considered as a single schedulable entity subject to a "group" deadline. Formally, we define the *group deadline* for the subtasks $T_i, \ldots, T_j$ to be $d(T_j)$ if $b(T_j) = 0$, and $d(T_j) + 1$ if $|w(T_{j+1})| = 3$. Intuitively, if we imagine a job of T in which each subtask is scheduled in the first slot of its window, then the remaining empty slots exactly correspond to the group deadlines of T. For example, in Fig. 2, T has group deadlines at slots $t + 3$, $t + 7$, and $t + 10$.

We let $D(T_i)$ denote the group deadline of subtask T_i. Formally, if T is heavy, then $D(T_i) = (\textbf{min}\ u :: u \geq$

$d(T_i)$ and u is a group deadline of T). For example, in Fig. 2, $D(T_1) = t + 3$ and $D(T_6) = t + 10$. If T is light, then $D(T_i) = 0$. For intra-sporadic task systems, the group deadline of a subtask is calculated assuming that all future subtasks of the same task are released as early as possible. Group deadlines can be calculated using a simple formula, though we omit it here due to space constraints.

Having explained the notion of a group deadline, we can now state the PD^2 priority definition.

PD^2 Priority Definition: Subtask T_i's priority at slot t is defined to be $(d(T_i), b(T_i), D(T_i))$, if it is eligible at t. Priorities are ordered using the following relation.

$$(d', b', D') \preceq (d, b, D) \equiv [d < d'] \vee [(d = d') \wedge (b > b')] \vee \ [(d = d') \wedge (b = b') \wedge (D \geq D')]$$

T_i's priority is at least U_j's at time t if both are eligible and $(d(U_j), b(U_j), D(U_j)) \ \preceq \ (d(T_i), b(T_i), D(T_i))$. $\square$

Thus, PD^2 prioritizes subtasks in deadline order. If T_i and U_j have equal deadlines, but $b(T_i) = 1$ and $b(U_j) = 0$, then the tie is broken in favor of T_i. This is because the window of T_i may overlap that of T_{i+1} (and hence *not* scheduling it may reduce the number of slots available to T_{i+1} by one, constraining the future schedule). If T_i and U_j have equal deadlines and b-bits, then their group deadlines are inspected to break the tie. If one is heavy and the other light, then the tie is broken in favor of the heavy task (by the definition of the group deadline). If both are heavy and their group deadlines differ, then the tie is broken in favor of the one with the later group deadline. Note that the subtask with the later group deadline can force a longer cascade of scheduling decisions in the future. Thus, choosing to schedule such a subtask early places fewer constraints on the future schedule. Any ties not resolved by PD^2 can be broken arbitrarily.

3 Scheduling of Aperiodic Tasks

An aperiodic task T is characterized by two integer parameters: its *release time* (or arrival time) $T.r$ and its *worst-case execution time* $T.e$. $T.r$ is usually not known before T arrives. Hard aperiodic tasks also have a deadline parameter $T.d$; for such tasks, $T.e$ and $T.d$ must be known upon arrival. For simplicity, we will assume that all aperiodic tasks in a system are either hard or soft, *i.e.*, both kinds are not simultaneously present.

On a uniprocessor, aperiodic tasks can be serviced by a single server task (*e.g.*, see [8, 9, 13, 16]). However, on a multiprocessor, the available processor capacity may exceed one, in which case it cannot be completely utilized by a single server unless it is scheduled in parallel. Even if the spare capacity is at most one, the available parallelism might be more fully exploited by using multiple servers. Nonetheless, in this section, we assume that there is just a single server. We postpone the consideration of multiple servers to Sec. 4. We further assume that the total utilization of the system including the aperiodic server is M, the number of processors. In other words, we assume that the weight of the aperiodic server is $M - \sum_{T \in \tau} wt(T)$ and that this weight is at most one. Note that, with a single server, it is sufficient to have a single global queue for the aperiodic tasks.

Whenever the aperiodic server gets scheduled, it selects an aperiodic task for execution if the queue is not empty. Hard aperiodic tasks are scheduled according to the *earliest-deadline-first* (EDF) policy, while soft aperiodic tasks are scheduled on a *first-come first-serve* (FCFS) basis. The real-time intra-sporadic tasks in the system are scheduled in a Pfair manner using the PD^2 algorithm (*i.e.*, "late" subtask releases are allowed for such tasks, but "early" releases are not). It is not necessary that the real-time tasks be scheduled in a Pfair manner; however, scheduling them in an ERfair manner would probably increase aperiodic response times, because allowing subtasks of the real-time tasks to execute early can delay the execution of the aperiodic server.

In the following subsection, we describe a generic admission-control test for hard aperiodic tasks. In later subsections, we describe the two kinds of servers considered in this paper: Pfair servers and ERfair servers. We also discuss how to evaluate worst-case response times for aperiodic tasks scheduled using these servers.

3.1 Admission-control Test for Hard Aperiodic Tasks.

We assume a generic *response-time function* $R\colon \mathcal{N} \mapsto \mathcal{N}$, *i.e.*, $R(e)$ gives the worst-case response time for e units of execution (by any number of aperiodic tasks). In other words, starting at time t, the server is guaranteed to complete e execution units by time $t + R(e)$ (*i.e.*, no latter than *slot* $t + R(e) - 1$). The actual evaluation of the response-time function depends on the server implementation, so we postpone consideration of it until later.

Suppose that, at time t, k aperiodic tasks are released. Let A denote the set of these tasks and let $d'_i, 1 \leq i \leq k$, be their deadlines. Assume that $d'_{i+1} \geq d'_i$ for $1 \leq i < k$. Merge this list of deadlines with the list of deadlines of aperiodic tasks admitted previously. Let this merged list be $d_1, d_2, \ldots, d_n$, where n is the total number of aperiodic tasks including the newly-released ones, and $d_{i+1} \geq d_i$ for $1 \leq i < n$. Let $T^{(i)}$ be the task with deadline d_i. Let e_i be the remaining execution time for $T^{(i)}$. Note that e_i may be less than the total execution time that was requested by $T^{(i)}$ because it may already have executed for some time before t. The algorithm shown in Fig. 3 can then be used as an admission-control test.

The idea behind the algorithm is as follows. First, note that all tasks admitted prior to the current time have been guaranteed to meet their deadlines. Thus, in lines 1–3, C (the set of accepted tasks) is initialized to include all previously-accepted tasks with a deadline at most d'_1 (the earliest deadline among all tasks in A). In lines 4–16, the remaining aperiodic tasks are considered in deadline order.

At each step, we do the following. Let the current task being considered be $T^{(i)}$. Tentatively admit this task and then check whether its deadline can be guaranteed (lines 5 and 6). If not, then there are two cases to consider. If $T^{(i)} \in A$ (refer to line 6), then T is rejected. Otherwise, we reject some other task in A that has already been added to C. (Such a task exists because $T^{(i)} \notin A$ is guaranteed to meet its deadline if no new tasks are admitted.) We repeat this rejection process until the deadline of $T^{(i)}$ can be guaranteed. Note that, in line 11, the tasks are rejected in order of non-increasing execution times. However, this is not necessary, and any other parameter indicating the "importance" of the newly-admitted tasks may be used.

The time complexity of the above procedure is $O(k \log k + (n + k)f)$, where f is the time complexity of the response-time function. In particular, sorting the newly-released tasks in deadline order takes $O(k \log k)$ time. Merging the two lists of deadlines takes $O(n)$ time. (Actually, this can be reduced to $O(\log n)$ time using more

```
1:  $C := \{T^{(i)} \mid d_i \le d'_1$ and $T^{(i)} \notin A\}$;
2:  $E := \sum_{T^{(i)} \in C} e_i$; $WCR := R(E)$;
3:  Let $T^{(j)}$ be the task with the largest index in $C$;
4:  for $i = j + 1$ to $n$ do
5:      $C := C \cup \{T^{(i)}\}$; $E := E + e_i$; $WCR := R(E)$;
6:      if $(t + WCR > d_i) \wedge (T^{(i)} \in A)$ then
7:          reject task $T^{(i)}$;
8:          $C := C - \{T^{(i)}\}$; $E := E - e_i$; $WCR := R(E)$
9:      else
10:         while $(t + WCR > d_i)$ do
11:             Let $e_r = \mathbf{max}\{e_l \mid T^{(l)} \in C \cap A\}$;
12:             reject task $T^{(r)}$;
13:             $C := C - \{T^{(r)}\}$; $E := E - e_r$; $WCR := R(E)$
14:         od
15:     fi
16: od
```

Figure 3: A generic admission-control procedure. The aperiodic tasks are considered in order of non-decreasing deadlines. At any point in the execution of the algorithm, the set C represents the aperiodic tasks that have already been considered. Note, however, that some of the tasks in $C \cap A$ might be rejected at some later point in the execution of this algorithm. E represents the total remaining execution time of the tasks in C. WCR represents the worst-case response time of the tasks in C. Task $T^{(r)}$ represents the task chosen for rejection and e_r is its execution time.

advanced data structures; however, the **for** loop requires $O(n)$ time anyway.) Note that the **max** calculation (line 11) is done at most k times. This is because each such calculation is followed by a rejection of a task in A. Thus, the **while** loop in line 10 iterates at most k times. Therefore, the number of evaluations of R is $O(n+k)$. Hence, the time complexity of the **for** loop is at most $(n+k)f$.

Correctness. At the end of the procedure, the set C represents all the admitted aperiodic tasks in the system. Note that, for each task in C, the worst-case completion time $(t + WCR)$ is at most its deadline; otherwise, some task in $C \cap A$ is removed until this holds (refer to lines (6) and (10)). Thus, every task in C is guaranteed to meet its deadline. Note also that a task that is admitted is never rejected at a later time.

3.2 Pfair Servers

In this approach, the aperiodic server is scheduled in a Pfair manner, *i.e.*, each subtask of the server is scheduled in its PF-window. Whenever the server gets scheduled by PD^2, it schedules some unfinished aperiodic task. If no such task exists, then the server has three options, which are described below. Let t be such a time, let S denote the server task, and let S_i denote its eligible subtask at time t. Fig. 4 illustrates the difference between the three variants using the following task set scheduled on two processors: a set Y consisting of four tasks of weight 1/4, and a set Z consisting of 22 tasks of weight 1/32. The server task S has weight $2 - 4 \times 1/4 - 22 \times 1/32 = 5/16$. A single aperiodic task A is released at time 2 with an execution requirement of 2 time units. Thus, the server has no task to schedule until time 2. The three options are as follows.

- *Idle* the processor. This is the simplest and most efficient scheme, but unnecessarily wastes processor time. (The main reason for introducing it is to compare the relative performance of the next two servers.)

  This variant is illustrated in Figure 4(a). At time 0, the subtask of S has the highest priority. Since the queue is empty at time 0, S just idles the processor. Note that only one task is scheduled in that slot.

- *Drop* S_i (*i.e.*, mark it as permanently ineligible) and select some other eligible subtask with lower priority. This variant does not waste processor time, but, as our experiments confirm, a simplistic implementation of it for multiple servers can produce worse results than either of the other schemes. Note that the overhead involved here is more than idling due to the need to reschedule some other task. In the worst case, the scheduler might have to make $M + s$ scheduling decisions, where s is the number of servers.

  Consider the example in Fig. 4(b). Note that there are two tasks scheduled in slot 0 as opposed to one in the idling variant case. (However, there is no improvement in the response time of A.)

- *Stall* the release of S_i until the next slot and declare that S has no eligible subtask at t. In the previous two schemes, the server is implemented as a periodic task, while here, it is implemented as an intra-sporadic task. More precisely, if the stalling server is scheduled when the aperiodic task queue is empty, it withdraws its current subtask S_i (*i.e.*, S_i is viewed as having not yet been released) and changes the release time of S_i to be the next slot. By delaying S_i's release, its deadline is also delayed and hence its priority relative to other subtasks drops. This method defers the release of S_i to the last moment possible (but, as the example below shows, S_i may be released *before* the next aperiodic task arrives). Intuitively, this scheme should perform better than the previous two schemes and the experiments confirm this. As in the previous scheme, the scheduler here might have to make $M + s$ scheduling decisions.

  Fig. 4(c) depicts an example execution of a stalling server. At time 0, S postpones the release of its next subtask until the next time slot, *i.e.*, slot 1. However, at time 1, there are two subtasks with priority higher than S's subtask; hence, S is not scheduled in slot 1. For this reason, the release of S's subtask is not further postponed. In the next time slot, S gets scheduled and it schedules A in that slot. Note that the response time is much better in this variant.

In each of the schemes above, the real-time tasks are guaranteed to meet their deadlines because of the optimality of PD^2. Note that none of these schemes requires that execution times be known: until a task completes execution, it will remain in the queue and the server will continue to execute it. (Obviously, for hard aperiodic tasks, execution times *are* needed for admission control.) For similar reasons, non-integral execution times pose no problems. When the server finishes executing a task, it selects the next task for execution. It does not really matter whether this switch happens at the end of a quantum or in the middle of a quantum. However, in the response-time calculations given below, we do assume integral worst-case execution costs, as before.

Calculation of the response-time function. We now present a constant-time procedure for evaluating $R(e)$, *i.e.*, the worst-case response time for e time units of execution when using a single Pfair server. We first

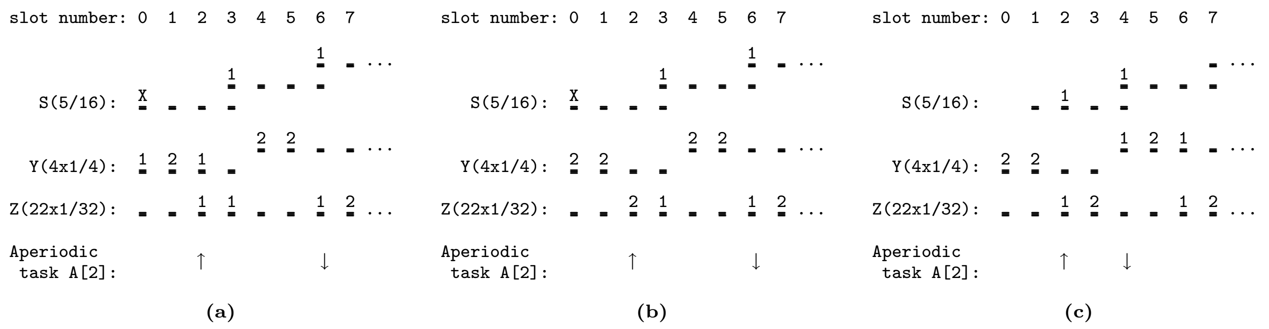


Figure 4: A partial schedule for a task set is shown under the different Pfair server variants. An 'X' in slot t refers to the situation where the Pfair server S is scheduled but the aperiodic task queue is empty. The aperiodic task is shown on a different line. The value in brackets indicates its execution time. The up-arrow and down-arrow denote its release time and finish time, respectively. **(a)** The idling variant. **(b)** The dropping variant. **(c)** The stalling variant.

define a new term. Subtask T_i is said to be *active* at time t if $d(T_i) \geq t$, T_i has not been scheduled yet, and T_{i-1} (if it exists) has already been scheduled. Note that since T_{i-1} has been scheduled, this also implies that

$$(i > 1 \Rightarrow t > r(T_{i-1})) \wedge (i = 1 \Rightarrow t \geq 0). \tag{9}$$

Let S denote the server task, assume the response-time function is called at time t, and let S_i be the active subtask of S at time t. For example, in Fig. 4, t is 2 (when task A arrives) and i is 2, 2, and 1, respectively, for the idling, dropping, and stalling variants. (Recall that "time 2" is the beginning of slot 2; thus, the response-time function is being called at the beginning of slot 2 and A potentially could be scheduled in this slot.)

Claim 1 *For the idling and the dropping variants of the Pfair server,* $R(e) \leq \left\lceil \frac{e+1}{wt(S)} \right\rceil$.

Proof: Note that S_{i+e-1} is the e^{th} subtask from time t. By the optimality of the PD^2 algorithm, S_{i+e-1} meets its deadline. Hence, by (3), $t+R(e) \leq \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - 1 + 1$. (The '+1' term is needed because the deadline in (3) refers to slots, not time.) Therefore, $R(e) \leq \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - t$. There are two cases to consider now depending on whether i is 1. If $i = 1$, then $t \geq 0$ (by (9)). In this case, $R(e) \leq \left\lceil \frac{1+e-1}{wt(S)} \right\rceil \leq \left\lceil \frac{e+1}{wt(S)} \right\rceil$ as required. On the other hand, if $i > 1$, then by (9), $t > r(S_{i-1})$, *i.e.*, $-t < -\left\lfloor \frac{i-2}{wt(S)} \right\rfloor$ (refer to (2)). Hence, we have $R(e) < \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - \left\lfloor \frac{i-2}{wt(S)} \right\rfloor$. It is easy to show that the right-hand side of this inequality is at most $\left\lceil \frac{e+1}{wt(S)} \right\rceil + 1$. □

Claim 2 *For the stalling variant of the Pfair server,* $R(e) \leq \left\lceil \frac{e}{wt(S)} \right\rceil + 1$.

Proof: In this case, we have $t \geq r(S_i)$. Note that t can be greater than $r(S_i)$; for example, at time $t = 2$ in Fig. 4(c), S_1 is active and $r(S_1) = 1$. Reasoning as in Claim 1, but this time using (7), we have $t + R(e) \leq r(S_{i+e-1}) + \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - \left\lfloor \frac{i+e-2}{wt(S)} \right\rfloor - 1 + 1$. Using $t \geq r(S_i)$, we get $R(e) \leq r(S_{i+e-1}) + \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - \left\lfloor \frac{i+e-2}{wt(S)} \right\rfloor - r(S_i)$. Subtasks S_i and S_{i+e-1} have the same offset with respect to their counterparts in a periodic system, since the server has e units to execute at time t. Let Δ denote this offset. Then, by (2), $r(S_{i+e-1}) = \left\lfloor \frac{i+e-2}{wt(S)} \right\rfloor + \Delta$ and $r(S_1) = \left\lfloor \frac{i-1}{wt(S)} \right\rfloor + \Delta$. Simplifying, we get $R(e) \leq \left\lceil \frac{i+e-1}{wt(S)} \right\rceil - \left\lfloor \frac{i-1}{wt(S)} \right\rfloor$, which implies $R(e) \leq \left\lceil \frac{e}{wt(S)} \right\rceil + 1$. □

Note that these bounds can be evaluated in $O(1)$ time independently of the real-time tasks in the system.

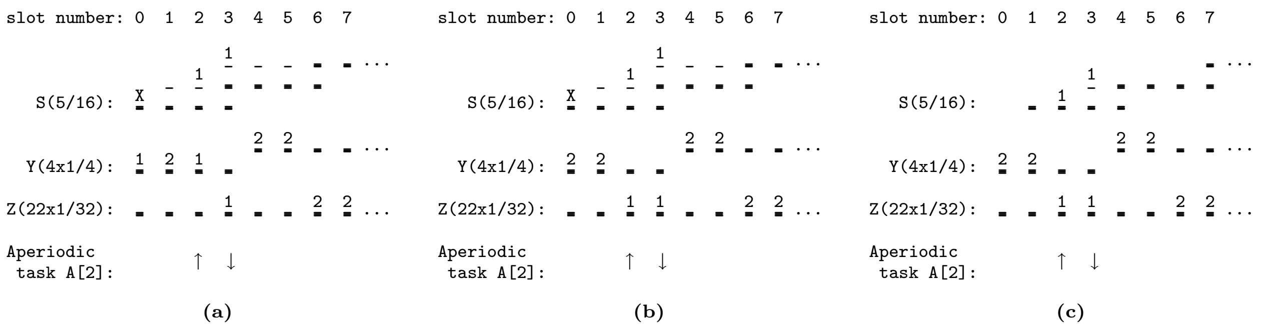


Figure 5: A partial schedule for a task set is shown under the different ERfair server variants. **(a)** The idling variant. **(b)** The dropping variant. **(c)** The stalling variant.

3.3 ERfair Servers

In this approach, the aperiodic server is scheduled in an ERfair manner. As mentioned earlier, PD^2 is optimal here as well [3, 17]. Hence, all the real-time intra-sporadic tasks meet their deadlines. Since the subtasks are allowed to execute before their windows, the rate at which an ERfair server serves the aperiodic tasks can be higher than that of a Pfair server. Hence, in general, aperiodic response times will be better when using an ERfair server as compared to a Pfair server. (On the other hand, ERfairness introduces more jitter into the server execution. Therefore, Pfair servers will be better for systems in which jitter is also a concern.) As before, we can have three different implementations of this server. Intuitively, the stalling scheme for ERfair servers should outperform all the other servers and experiments confirm this.

Fig. 5 illustrates the three variants for the example task set considered in Fig. 4. Inset (a) illustrates the idling variant: when the server gets scheduled in slot 0, it simply idles the processor. Note that because of the ERfair nature of the server, the server is able to schedule task A in slot 2. The behavior of the dropping variant (inset (b)) is similar. In the stalling variant (inset (c)), because the second subtask is early-released, A finishes by time 4 as opposed to time 5 in the stalling variant of the Pfair server (refer to Fig. 4(c)).

One interesting point about the stalling scheme for the ERfair server is the manner in which stalling and ERfairness complement each other. Note that, when the server is backlogged (*i.e.*, the queue is non-empty), stalling is not used and the ERfair nature of the server gives good response times. On the other hand, when not backlogged, as noted earlier, the stalling scheme provides the best response time among the three schemes.

Calculation of the response-time function. The response-time calculation given for Pfair servers applies to ERfair servers as well, but it is not very accurate in this case. The following procedure gives us a more accurate value in the ERfair case. As before, let S denote the server and let t be the time when the response-time function is called. Also, let i be the subtask of S active at t and let $d = d(S_{e+i})$. (Thus, $e+1$ subtasks of S are guaranteed to execute in the interval $[t, d]$.) The response time $R(e+1)$ is calculated in a recursive manner using the value for $R(e)$. Because $R(e)$ represents the worst-case response time for e time units, the subtask S_{e+i-1} is guaranteed to complete by time $t + R(e)$. Thus, S_{e+i} is eligible for execution at or after $t + R(e)$.

Let set H consist of all subtasks of all tasks that can contend with S_{e+i} in the interval $[t + R(e) + 1, d]$ and that have higher priority. These are the only subtasks that can delay the execution of S_{e+i} after time $t + R(e)$. Let $h = |H|$. We now bound $R(e + 1)$ in terms of $R(e)$ and h and then describe how to evaluate h.

Claim 3 *For the ERfair server S, $R(e + 1) \leq R(e) + \lfloor \frac{h}{M} \rfloor + 1$.*

Proof: Note that the subtasks in H can execute for at most $\lfloor \frac{h}{M} \rfloor$ time without leaving a processor idle. Therefore, because S_{e+i} is eligible to execute at or after $t + R(e) + 1$, S_{e+i} is guaranteed to execute in the interval $[t + R(e) + 1, t + R(e) + \lfloor \frac{h}{M} \rfloor + 1]$. □

Claim 4 *The number of subtasks of* periodic *task U in H is at most $\lfloor (d+1) \cdot wt(U) \rfloor - \lfloor (t+R(e)+1) \cdot wt(U) \rfloor$.*

Proof: Note that, due to space limitations, we have limited this claim to pertain only to *periodic* tasks; if U is intra-sporadic (*i.e.*, releases some subtasks late), then the needed bound is at most one more than that stated in the claim. (Late releases can introduce at most one additional PF window over an interval.)

Recall that all real-time tasks are scheduled in a Pfair manner. Hence, the number of subtasks of U in H is equal to the number of subtasks of U that have a deadline in the interval $[t + R(e) + 1, d]$.

Let U_k be the latest subtask of U satisfying $d(U_k) \leq d$. In other words, no task after U_k is in H. Also, let U_j be the earliest subtask of U satisfying $d(U_j) \geq t + R(e) + 1$, *i.e.*, no task before U_j can be in H.

Note that subtasks $U_j, \ldots, U_k$ *may* contend with S_{e+i} and also *may* have higher priority. (Whether they actually have higher priority may depend on PD^2's tie-breaks.) Thus, the number of subtasks of U that belong to H is at most $k - j + 1$. Using (3), we can obtain a formula for k as follows.

$$d(U_k) \leq d \;\Leftrightarrow\; \left\lceil \frac{k}{wt(U)} \right\rceil - 1 \leq d \;\Leftrightarrow\; \frac{k}{wt(U)} \leq d + 1$$

Because we are interested in the largest value of k, it follows that $k = \lfloor (d+1) \cdot wt(U) \rfloor$. Similarly, we can obtain a formula for j as follows, where x denotes $t + R(e) + 1$.

$$d(U_j) \geq x \;\Leftrightarrow\; \left\lceil \frac{j}{wt(U)} \right\rceil - 1 \geq x \;\Leftrightarrow\; \left\lceil \frac{j}{wt(U)} \right\rceil > x \;\Leftrightarrow\; \frac{j}{wt(U)} > x$$

Therefore, $j = \lfloor x \cdot wt(U) \rfloor + 1$. The required result follows. □

Note that, given the value of $R(e)$, the expression in Claim 4 can be calculated in constant time, and hence, h can be calculated in time linear in the number of real-time tasks in the system. This calculation can be improved to take into account the b-bit and group deadline of the PD^2 priority definition without increasing asymptotic complexity; however, we do not describe this extension here because of space limitations.

To obtain the actual value of $R(e)$, we take the minimum of the value obtained from Claim 3 and from Claim 1 or 2. In general, the value obtained by the above procedure should be smaller. As an example, consider the task set in Fig. 5. For all variants, when the aperiodic task is released at time 2 and response times are calculated using Claim 3, $R(1) = 1$ and $R(2) = 4$, *i.e.*, the aperiodic task is guaranteed to complete before time

6. In fact, it actually completes by time 4, as seen in the figure. The response time estimated using Claims 1 and 2 are 10 ($\lceil\frac{3\times16}{5}\rceil$) and 8 ($\lceil\frac{2\times16}{5}\rceil$+1), respectively. Thus, this new approach provides a better estimate.

Although the above procedure is more accurate, its time complexity is $O(Ne)$, where N is the number of real-time tasks in the system. In contrast, the earlier procedure runs in constant time. Thus, there is a trade-off between efficiency and accuracy. A simplistic implementation of the ERfair admission-control test requires $O((n+k)Ne + k\log k)$ time, where n, e, and k are as defined in Sec. 3.1. Note, however, that when calculating $R(e)$, we also calculate $R(i)$, where $1 \leq i < e$. These values can be stored to avoid recomputing later. If this is done, the admission-control test completes in $O(Ne + n + k + k\log k) = O(Ne + n + k\log k)$ time.

4 The Multiple Server Case

In many real-time systems, the spare processor capacity might exceed one. In the scheduling schemes that we are considering, a task's utilization is at most one. Hence, multiple aperiodic servers may be needed to use all the spare capacity. In this section, we consider several issues pertinent to systems with multiple servers. Before continuing, we state one property about Pfair and ERfair scheduling that is used below [4].

Lemma 1 *A periodic task of weight w receives at least $\lfloor w \cdot t \rfloor$ units of processor time in the interval* $[0, t]$.

In the paragraphs that follow, we discuss some heuristics for determining the number of servers and their weights. We then discuss the relative merits of global versus per-server queues and briefly describe an admission-control test for hard aperiodic tasks. Finally, we discuss an approach to better utilize the available parallelism.

Number of servers and weight distribution. Suppose the spare processor capacity is distributed by creating s servers. Let $w_i, 1 \leq i \leq s$, be the weights of these servers, where $0 < w_i \leq 1$. Also, let $l+x = \sum_{i=1}^{s} w_i$, where l is an integer and $0 \leq x < 1$. Thus, $l + x$ is the total spare processor capacity that the servers can consume. For any t, $\sum_{i=1}^{s} w_i \cdot t = l \cdot t + x \cdot t$. Now,

$$\sum_{i=1}^{s} \lfloor w_i \cdot t \rfloor \leq \left\lfloor \sum_{i=1}^{s} w_i \cdot t \right\rfloor = \lfloor l \cdot t + x \cdot t \rfloor = l \cdot t + \lfloor x \cdot t \rfloor$$

Thus, by Lemma 1, the service guaranteed to s servers by time t is at most that guaranteed to $l+1$ servers of weights $x, 1, 1, \ldots, 1$. Hence, better worst-case response times result using $l+1$ servers with l servers of unit weight. We call this the *greedy* policy. It is important to note that the analysis given here is only a guideline: actual response times may sometimes be better in the s-server case because of the extra parallelism available.

Global versus per-server queues. The strategy here varies depending on whether the aperiodic tasks are hard or soft. In the case of soft aperiodic tasks, where we are more interested in improving average response times, it is better to have a global queue rather than partition. This is because, with partitioning, a server might be scheduled when its own queue is empty and ready aperiodic tasks are waiting in other queues.

In the case of hard aperiodic tasks scheduled using the EDF algorithm, it is better to partition. In the global-queue case, using EDF can result in arbitrarily low utilization, as shown in [10]. The example presented there involves three tasks with (execution-requirement, deadline) parameters of $(2, p)$, $(2, p)$, and $(p, p+1)$, respectively, where $p \geq 4$. Note that this task set can be easily scheduled on two processors using partitioning. However, global EDF does not schedule this task set correctly. This exact situation arises when there are two server of weight 1 each and all three tasks arrive together. One way to alleviate this problem is for the servers to use Pfair scheduling techniques (rather than EDF) when scheduling hard aperiodic tasks. However, the servers themselves may execute at different rates, in which case the problem is similar to a *uniform* rather than an *identical* multiprocessor problem, and no optimal solution is known in the uniform case.

Admission control for hard aperiodic tasks. We only consider the partitioning case here. In this case, one of the admission-control tests proposed for the single-server case can be used. The only question remaining is how to partition the aperiodic tasks. One approach is to use traditional bin-packing algorithms such as first-fit or best-fit. Note, however, that the bin-packing algorithms need to be quite fast. Overall time complexity will depend on the time complexity of the admission-control test. If we use the response-time functions of Claims 1 and 2, then the admission-control test takes $O(n_i)$ time for every newly-released aperiodic task, where n_i is the number of tasks assigned to server i. Hence, the bin-packing assignment can be done in $O(\sum_i n_i) = O(n)$ time per aperiodic task, where n is the total number of aperiodic tasks (assigned to all servers).

Background servers. The intra-sporadic model allows late subtask releases. If the real-time tasks in the system frequently release subtasks late, then processors may often be idle (even in a system that is heavily utilized in a worst-case sense). Idle slots may also result if a job of a real-time task finishes earlier than its estimated worst-case execution time. Note that if an aperiodic server has tasks to schedule during a potentially idle time slot, it will do so. However, if the number of idle processors exceeds the number of eligible servers, and is less than number of unfinished aperiodic tasks, then processor time will be unnecessarily wasted. This is illustrated by the following example.

Consider a two-processor system on which two sets of intra-sporadic tasks are scheduled: a set A of four tasks of weight $1/4$, and a set B of seven tasks of weight $1/16$. In addition, there is one aperiodic server with weight $2 - 4(1/4) - 7(1/16) = 9/16$. Fig. 6 shows the PD^2 schedule that results assuming four aperiodic tasks with execution times 3, 2, 2, and 1 are released at times 0, 1, 10, and 10, respectively. The budget graph shows the remaining execution time of the aperiodic tasks. From the figure, we can see that the aperiodic tasks finish at times 4, 7, 12, and 13, respectively. Note that the fourth aperiodic task is serviced only after the completion of the third aperiodic task at time 12. Also note that a processor is idle in the interval $[10, 11]$. Thus, the fourth aperiodic task could actually have run in parallel with the third in slot 10.

One way to utilize this extra processor time is to have a set of background servers, one on each processor. Such a server is scheduled if its processor becomes idle while unfinished aperiodic tasks exist. Using background servers results in better utilization of the available parallelism than is otherwise possible.

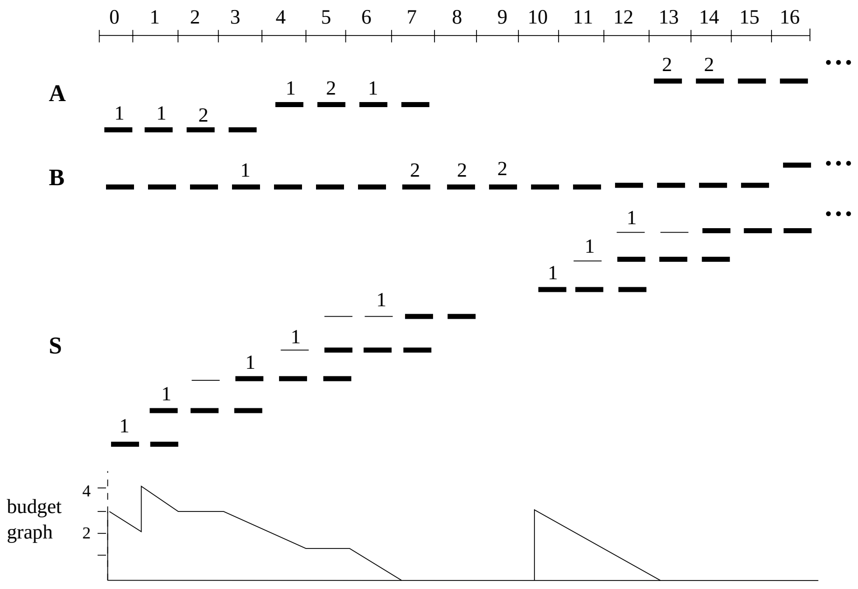


Figure 6: A and B are two sets of real-time intra-sporadic tasks and S represents the intra-sporadic task used as a stalling ERfair server. The budget graph illustrates the remaining execution of the aperiodic tasks.

5 Performance Studies

We conducted simulations to compare our proposed servers with background scheduling under various conditions. Performance was measured by computing the average aperiodic response time as a function of the utilization of the periodic tasks (*i.e.*, the spare capacity). Each aperiodic task's response time has been normalized with respect to its execution cost, *e.g.*, a value of 3 indicates a response time that is three times a task's execution cost. Before explaining the results that were obtained, we explain the experimental setup.

There are a number of factors that affect the scheduling of aperiodic tasks on multiprocessors, among them, the number of processors, the total utilization of the periodic tasks, and the distribution of the aperiodic task arrivals. We varied these three parameters to determine the effect on response times. In each experiment, the number of servers and weights was determined using the greedy policy described in Sec. 4. Also, a single global queue was maintained for the aperiodic tasks.

Forty different periodic task sets having the same total utilization were generated, along with forty different aperiodic task sets, and simulations were conducted for each combination of these task sets. Thus, each point in each graph is the result of 1,600 simulations. Each line in each graph was obtained by 40 such points uniformly distributed in the interval $[M/2, M - 0.25]$, where M is the number of processors. In other words, the spare capacity was varied from 0.25 to $M/2$ and the effect on the response times was then determined. We conducted these experiments for 2, 4, 8, 16, and 32 processors. We also studied the effect on the servers' performance that results from changing the distribution of the aperiodic arrivals from uniform to bursty. However, due to lack of space, we are able to present only some of the graphs here. Fortunately, the general trend seen in these graphs applies to the others as well (which also indicates the scalability of our approaches).

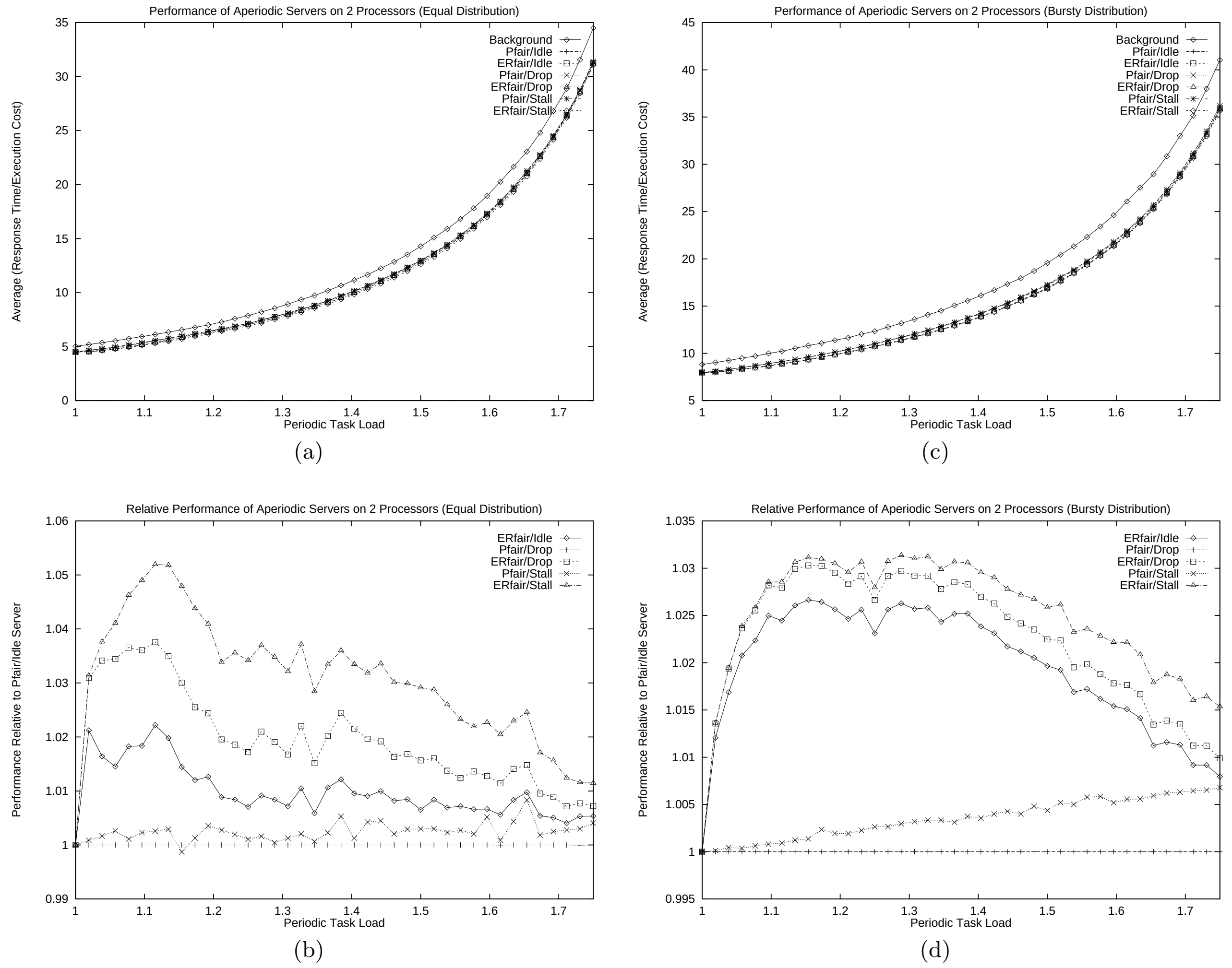
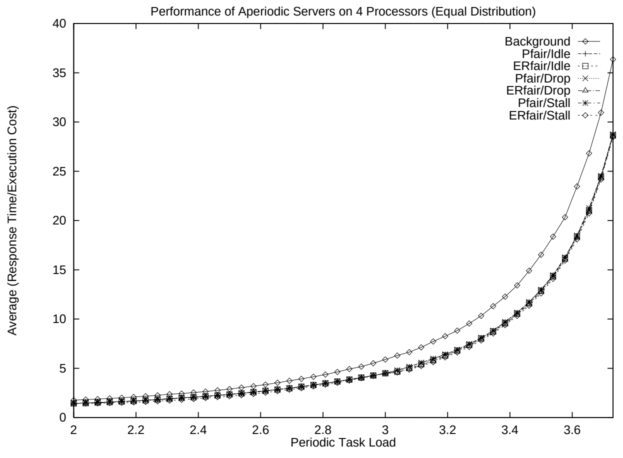


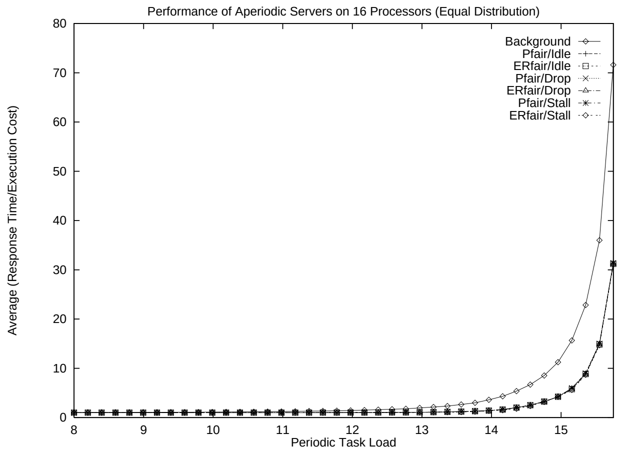
Figure 7: Simulation results on two processors. **(a)** & **(b)** Uniform distribution of aperiodic task releases. **(c)** & **(d)** Bursty releases. **(a)** and **(c)** show the actual performance (*i.e.*, normalized response times) of each scheme. In **(b)** and **(d)**, performance relative to an idling Pfair sever is illustrated.

Results. The graphs in Figs. 7 and 8 illustrate the performance of the six servers described in this paper. Insets (a) and (c) in both figures compare the performance of these servers to background scheduling, and insets (b) and (d) demonstrate the performance of these servers relative to an idling Pfair server (which is the most simplistic server, and thus an obvious choice for a baseline measurement).

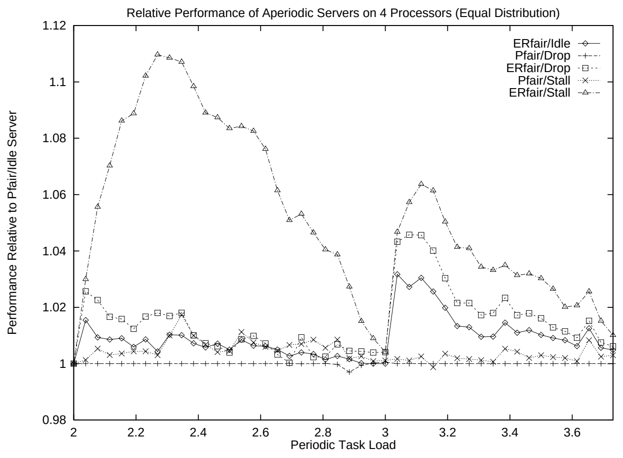
Fig. 7 illustrates the performance of the servers on two processors. There is one server here, whose weight is varied from 0.25 to 1. In insets (a) and (b), the aperiodic task releases are uniformly spaced. As can be easily seen, all the server schemes give better performance than background scheduling; as the periodic task load increases, the improvement in (a) approaches 9%. As expected, the stalling variant of the ERfair server provides the best performance; as seen in (b), its performance is up to 5% better than the idling Pfair server. In insets (c) and (d), the aperiodic task releases are bursty and grouped together near time zero. Similar results are seen here, although the improvement over background scheduling is a bit better (approaching about 17%). Note that, in this case, the server is continuously backlogged and hence the differences in inset (d) are less



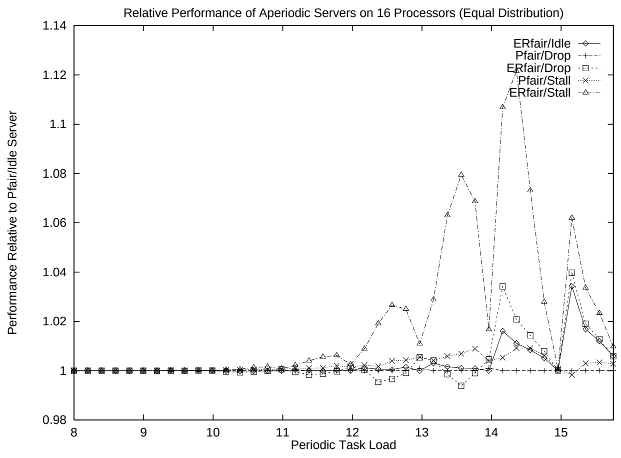
(a)



(c)



(b)



(d)

Figure 8: Simulation results on four and sixteen processors for uniform aperiodic tasks. **(a)** & **(b)** Four processors. **(c)** & **(d)** Sixteen processors. **(a)** and **(c)** show the actual values of the normalized response times. **(b)** and **(d)** illustrate the performance of the servers relative to the idling Pfair server.

pronounced. (Recall that the three variants considered earlier come into play when the servers are idle.)

Fig. 8 illustrates the performance of the servers on 4 and 16 processors with uniform aperiodic task arrivals. As seen in insets (a) and (c), the servers perform even better here compared to background scheduling, approaching 35% in (a) and 170% in (c). This illustrates the scalability of our approach as the number of processors increases. In fact, in experiments conducted for a 32-processor system (not shown here), the improvement over background scheduling approached 275% in the bursty case, and 500% in the uniform case. Note that the graphs in insets (b) and (d) are quite different from their counterparts in Fig.7. This is because the number of servers here might be more than one, and under the greedy policy, many of these servers will have a weight of one. Such a server is scheduled in every slot, and hence neither early-releasing nor stalling can improve its performance. When the periodic task load is integral, the greedy policy in fact results in *all* servers having a weight of one. This is why the various curves in insets (b) and (d) coincide at integral load levels. Intuitively, the difference between the implementations will be more pronounced when there are fewer unit-weight servers and when the non-unit-weight server's weight is around 0.5. When a server's weight is *too* small (much less than 0.5), its low utilization tends to keep the queue full, in which case the three variants perform similarly. Note that, in insets (b) and (d), ERfair/stall is up to 12% better than Pfair/drop (which is quite a bit more than in Fig. 7). This suggests that the differences among the various schemes we have proposed become more pronounced on larger systems. Note further that, in inset (d), ERfair/drop occasionally performs worse than Pfair/drop and Pfair/idle. This is because, when subtasks can become eligible earlier, there is a greater potential that a subtask is dropped (there are more slots where this could happen). Hence, response times suffer.

6 Conclusion

In this paper, we have presented two server implementations for multiplexing aperiodic and recurrent real-time tasks in fair-scheduled multiprocessor systems. This is the first paper to consider the problem of integrating support for aperiodic tasks within fair multiprocessor scheduling algorithms. The Pfair approach is similar to the uniprocessor constant utilization server [9], whereas the ERfair approach is similar to the uniprocessor total bandwidth server [16]. Note that the difference between these two approaches is the work-conserving nature of ERfair. We have also provided admission-control tests for the scheduling of hard aperiodic tasks, and have pointed out some additional complexities arising in server-based implementations on multiprocessors along with some ways to handle them. Most of these complexities arise because of the parallelism that exists in such systems. We have also provided experimental results that demonstrate the effectiveness of our implementations.

Although we have suggested techniques for handling many issues, a number of interesting problems remain. For example, we have assumed that both hard and soft aperiodic tasks are not simultaneously present. A simple way to integrate both is to use background scheduling for soft tasks. However, it should be possible to devise more sophisticated techniques that result in much better response times. Another topic that we did not consider in this paper is that of reclaiming unused server bandwidth when the aperiodic load is light. This is in fact done to some extent in the stalling scheme, at the price of postponing server subtask deadlines, which may not always be necessary. Another approach might be to allow subtasks of the real-time tasks to be released early when the aperiodic queue is empty. It would be interesting to analytically and empirically evaluate this strategy.

References

[1] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proc. of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

[2] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, Dec. 2000.

[3] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 76–85, June 2001.

[4] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[5] 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.

[6] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proc. of the Fourth ACM Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2000.

[7] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers. In *Proc. of the 7th IEEE Real-Time Technology and Applications Symposium*, May 2001.

[8] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10:1261–1269, Oct. 1989.

[9] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proc. of 9th Euromicro Workshop on Real-Time Systems*, pages 191–199, June 1997.

[10] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[11] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995.

[12] K. Jeffay and S. Goddard. The rate-based execution model. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 304–314, Dec. 1999.

[13] J. P. Lehoczky and S. Ramos-Thuel. An optimal algoririthm for scheduling soft-aperiodic tasks in fixed priority preemptive systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 110–123, Dec. 1992.

[14] J.P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. of IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[15] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[16] M. Spuri and G.C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[17] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Submitted to the 34th Annual ACM Symposium on Theory of Computing*, 2001.