Deadline Fair Scheduling: Bridging the Theory and Practice of Proportionate Fair Scheduling in Multiprocessor Systems *

Abhishek Chandra, **Micah Adler** and **Prashant Shenoy**

Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003
{abhishek,micah,shenoy}@cs.umass.edu

Abstract

Proportionate-Fairness (P-fairness) is a strict notion of proportional-share allocation defined for real-time systems, that generalizes easily to multiprocessor environments. In this paper, we present Deadline Fair Scheduling (DFS), a proportionate-fair CPU scheduling algorithm for multiprocessor servers. A particular focus of our work is to investigate practical issues in instantiating P-fair schedulers into conventional operating systems. We show via a simulation study that characteristics of conventional operating systems such as the separate scheduling of individual processors, arrivals and departures of tasks, and variable quantum durations can cause a P-fair scheduler such as DFS to become non-work-conserving. To overcome this drawback, we combine DFS with an auxiliary work-conserving scheduler to ensure work-conserving behavior at all times. We then propose techniques to account for processor affinities while scheduling tasks in multiprocessor environments. We implement the resulting scheduler in the Linux kernel and evaluate its performance using various applications and benchmarks. Our experimental results show that DFS can achieve proportional allocation, performance isolation and work-conserving behavior at the expense of a small increase in the scheduling overhead. We conclude that incorporating practical considerations such as work-conserving behavior and processor affinities into a P-fair scheduler such as DFS can result in a practical approach for scheduling tasks in a multiprocessor operating system.

1 Introduction

Recent advances in computing and communication technologies have led to a proliferation of demanding applications such as streaming audio and video players, multi-player games, and online virtual worlds. A key characteristic of these applications is that they impose (soft) real-time constraints, and consequently, require predictable performance guarantees from the underlying operating system. Several resource management techniques have been developed for predictable allocation of processor bandwidth to meet the needs of such applications [10, 14, 15]. Proportionate fair schedulers are one such class of scheduling algorithms [6]. A *proportionate-fair (P-fair)* scheduler allows an application to request x_i time units every y_i time quanta and guarantees that over any T quanta, $T > 0$, a continuously running application will receive between $\lfloor \frac{x_i}{y_i} \cdot T \rfloor$ and $\lceil \frac{x_i}{y_i} \cdot T \rceil$ quanta of service. P-fairness is a strong notion of fairness, since it ensures that, at any instant,

*An earlier version of this paper appeared in the Proceedings of the 21st IEEE Real-Time Technology and Applications Symposium (RTAS 2001), Taipei, Taiwan ROC, May 2001.

no application is more than one quantum away from its due share. Another characteristic of P-fairness is that it generalizes to environments containing multiple instances of a resource such as multiprocessor systems.

Several P-fair schedulers have been proposed over the past few years [1, 5, 17]. Most of these research efforts have focused on theoretical analysis of these schedulers. In this paper, we consider practical issues that arise when implementing a P-fair scheduler into a multiprocessor operating system kernel. Our research effort leads to several contributions. First, we propose *Deadline Fair Scheduling (DFS)*—a multiprocessor scheduling algorithm that is provably P-fair in the presence of certain system model assumptions that are also made by most existing P-fair algorithms. However, DFS has the added advantage of being well-defined even in the presence of typical characteristics of multiprocessor operating systems such as the asynchrony in scheduling multiple processors, arrivals and departures of tasks, and variable quantum durations. We then show using simulations that these characteristics can cause a P-fair scheduler such as DFS to become non-work-conserving, and hence, to lose its property of P-fairness (We prove that a P-fair scheduling algorithm is also work-conserving). Since a non-work-conserving scheduler also reduces the processor utilization by causing a processor to remain idle even in the presence of runnable tasks, an important practical consideration is to ensure work-conserving behavior at all times. To achieve this objective, we draw upon the concept of fair airport scheduling [12] to combine DFS with an auxiliary work-conserving scheduler in order to guarantee work-conserving behavior. Another practical consideration for multiprocessor schedulers is the ability to take *processor affinities* [25] into account while making scheduling decisions—scheduling a task on the same processor enables it to benefit from data cached from previous scheduling instances and improves the effectiveness of a processor cache. We propose techniques that enable a P-fair scheduler such as DFS to account for processor affinities; our technique involves a practical tradeoff between three conflicting considerations—fairness, scheduling efficiency, and processor cache performance.

We have implemented DFS in the Linux operating system and have made the source code available to the research community.[1] We chose Linux over a real-time kernel since we are primarily interested in examining the practicality of using a P-fair scheduler for multimedia and soft real-time applications and we believe that such applications will typically coexist with traditional best-effort applications on a conventional operating system. We experimentally evaluate the efficacy of our scheduler using various applications and benchmarks. Our results show that DFS can achieve proportional allocation, application isolation and work-conserving behavior, albeit at a slight increase in scheduling overhead. We conclude from these results that a careful blend of theoretical and practical considerations can yield a P-fair scheduler suitable for conventional multiprocessor operating systems.

The rest of this paper is structured as follows. Section 2 presents basic concepts in fair proportional-share scheduling. Section 3 presents our deadline fair scheduling algorithm. Sections 4 and 5 discuss two practical issues in implementing DFS, namely work-conserving behavior and processor affinities. Section 6 presents the details of the DFS implementation in Linux. Section 7 presents the results of our experimental evaluation and we conclude in Section 8.

[1] See http://lass.cs.umass.edu/software/gms.

2 Proportional-Share Scheduling and Proportionate-Fairness: Background

Popular applications such as streaming audio and video and multi-player games have timing constraints and require performance guarantees from the underlying operating system. Such applications fall under the category of soft real-time applications—due to their timing constraints, the utility provided to users is maximized by maximizing the number of real-time constraints (e.g., deadlines) that are met, but unlike hard real-time applications, occasional violations of these constraints do not result in incorrect execution or catastrophic consequences.[2]

Several resource management mechanisms have been developed to explicitly deal with soft real-time applications [4, 10, 11, 14, 15, 16, 18, 19, 26]. These mechanisms broadly fall under the category of *proportional-share schedulers*—these schedulers associate an intrinsic rate with each application and allocate bandwidth in proportion to the specified rates. One class of proportional-share schedulers are based on *generalized processor sharing (GPS)* [19]. GPS assumes that tasks can be serviced in terms of infinitesimally small time quanta, and hence GPS-fairness allocates CPU bandwidth to tasks in the proportion of their weights at all times. Practical instantiations of GPS such as weighted fair sharing [9, 20] and start-time fair queuing [11] provide looser bounds on how far tasks can be from their GPS shares at any time. *Proportionate-fair (P-fair)* schedulers are another class of proportional-share schedulers based on the notion of proportionate progress [6]. Under this notion, each application requests x_i quanta of service every y_i time quanta. The scheduler then allocates processor bandwidth to applications such that, over any T time quanta, $T > 0$, a continuously running application receives between $\lfloor \frac{x_i}{y_i} \cdot T \rfloor$ and $\lceil \frac{x_i}{y_i} \cdot T \rceil$ quanta of service. P-fairness is a strong notion of fairness, since it ensures that, at any instant, no application is more than one quantum away from its due share. Unlike GPS, P-fairness assumes that applications are allocated finite duration quanta (and thus is a more practical notion of fairness). In addition, it ensures tighter bounds on the possible unfairness than practical instantiations of GPS.

Several algorithms have been proposed that achieve P-fairness in an ideal model—synchronized, fixed quantum durations and a fixed task set [1, 5, 17]. In practice, however, these ideal conditions do not hold in real systems. Blocking or I/O events might cause an application to relinquish the processor before it has used up its entire allocated quantum, and hence, quantum durations tend to vary from one quantum to another. These variable quantum lengths also result in asynchronous scheduling of multiple processors in a multiprocessor system, i.e., each processor calls the scheduler independently, and hence, the scheduling of tasks on different processors is not simultaneous. Moreover, P-fairness implicitly assumes that the set of tasks in the system is fixed. In practice, arrivals and departures of tasks as well as blocking and unblocking events can cause the task set to vary over time.

Several recent research efforts have focused on relaxing some of the above assumptions of the ideal system model. Recently, conditions for task arrivals and departures have been derived for a P-fair algorithm that avoids any deadline misses [23]. The goal of our work is to allow any arrival/departure pattern in the system, even if it results in occasional deadline violations. In addition, the notion of P-fairness has been

[2]Multimedia/streaming media applications are an important subset of the class of soft real-time applications. Note that, there could be other applications such as virtual reality that are soft real-time but do not involve streaming audio and video.

generalized to other models of sporadic and non-periodic tasks [22, 24] and for soft real-time tasks [21]. The focus of our work is orthogonal to these efforts, as our goal is to exploit the notion of periodicity to provide proportional-share for tasks that may fit any kind of computational model. In addition, we also consider a system model with variable quantum lengths and asynchronous scheduling of multiple processors, that has not been considered by these research efforts.

In this paper, we propose an algorithm that achieves P-fairness in the presence of the above-mentioned ideal system assumptions. In addition, this algorithm is clearly defined even when the system has variable quantum durations and arrivals and departures of tasks. To seamlessly account for these non-ideal system conditions, in this paper, we use a modified definition of P-fairness for the ideal model: Let ϕ_i denote the share of the processor bandwidth that is requested by task i in a p-processor system. Then, over any T time quanta, $T > 0$, a continuously running application should receive between $\lfloor \frac{\phi_i}{\sum_j \phi_j} \cdot pT \rfloor$ and $\lceil \frac{\phi_i}{\sum_j \phi_j} \cdot pT \rceil$ quanta of service. Observe that, in the ideal model, this definition reduces to the original definition of P-fairness in the case where $\phi_i = \frac{x_i}{y_i}$ and $\sum_j \phi_j = p$ (which corresponds to the tasks using up all the quantums available on the processors).

Another dimension for classifying schedulers is whether they are work-conserving or non-work-conserving. A scheduler is defined to be work-conserving if it never lets a processor idle as long as there are runnable tasks in the system. Non-work-conserving schedulers, on the other hand, can let a processor idle even in the presence of runnable tasks. Intuitively, a work-conserving proportional-share scheduler treats the shares allocated to an application as lower-bounds—a task can receive more than its requested share if some other task does not utilize its share. A non-work-conserving proportional-share scheduler treats these shares as upper-bounds—a task does not receive more than its requested share even if a processor is idle. To achieve good resource utilization, schedulers employed in conventional operating systems tend to be work-conserving in nature. The notion of P-fairness has been extended to incorporated work-conserving behaviour [2, 3] that relaxes the upper bound on the CPU service received by a task. We consider alternate ways of enhancing P-fair schedulers to achieve work-conserving behavior.

In what follows, we first present our scheduling algorithm for multiprocessor environments based on the notion of proportionate-fairness. We then consider two practical issues that will require us to relax the notion of strict P-fairness (i.e, we trade strict P-fairness for more practical considerations).

3 Deadline Fair Scheduling

3.1 System Model

Consider a p-processor system that services N tasks. At any instant, some subset of these tasks will be runnable while the remaining tasks are blocked on I/O or synchronization events. Let n denote the number of runnable tasks at any instant. In such a scenario, the CPU scheduler must decide which of these n tasks to schedule on the p processors. We assume that each scheduled task is assigned a quantum duration of q_{max}; a task may either utilize its entire allocation or voluntarily relinquish the processor if it blocks before its allocated quantum ends. Consequently, as is typical on most multiprocessor systems, we assume that quanta on different processors are *neither synchronized with each other, nor do they have a fixed duration*. An important consequence of this assumption is that each processor needs to individually invoke the CPU scheduler when its current quantum ends, and hence, scheduling decisions on different processors are not synchronized with one another.

Given such an environment, assume that each task specifies a share ϕ_i that indicates the proportion of the processor bandwidth required by that task. Since there are p processors in the system and a task can run on only one processor at a time, each task cannot ask for more than $\frac{1}{p}$ of the total system bandwidth. Consequently, a necessary condition for feasibility of the current set of tasks is as follows:

$$\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \leq \frac{1}{p} \tag{1}$$

We refer to this condition as the *weight feasibility constraint*. This constraint can be maintained by employing admission control on task arrivals and departures, or using a weight readjustment algorithm [7]. Our Deadline Fair Scheduling (DFS) algorithm achieves allocations corresponding to these weights based on the notion of proportionate-fairness. To see how this is done, we first present the intuition behind our algorithm and then provide the precise details.

3.2 DFS: Key Concepts

Conceptually, DFS schedules each task periodically; the period of each task depends on its share ϕ_i. DFS uses an *eligibility criterion* to ensure that each task runs at most once in each period and uses *internally generated deadlines* to ensure that each task runs at least once in each period. The eligibility criterion makes each task eligible at the start of each period; once scheduled on a processor, a task becomes ineligible until its next period begins (thereby allowing other eligible tasks to run before the task runs again). Each eligible task is stamped with an internally generated deadline. The deadline is typically set to the end of its period in order for the task to run by the end of its period. DFS schedules eligible tasks in *earliest deadline first* order to ensure each task receives its due share before the end of its period. Together, the eligibility criterion and the deadlines allow each task to receive processor bandwidth based on the requested shares, while ensuring that no task gets more or less than its due share in each period. The following example illustrates this process.

Example 1 Consider a dual-processor system that services three tasks with shares $\phi_1 = 2$ and $\phi_2 = \phi_3 = 1$. This could correspond to the tasks asking for $(x_1, y_1) = (1, 1)$ and $(x_2, y_2) = (x_3, y_3) = (1, 2)$. The requested allocation can be achieved by running the first task continuously on one processor and alternating between the other two tasks on the other processor. We show how this can be done using periods and deadlines. The period of the first task can be set to 1 and that of the other two tasks to 2. Thus, task 1 becomes eligible every time unit, while tasks 2 and 3 become eligible every other time unit. Once eligible, a task is stamped with a deadline that is the end of its period. Once scheduled, a task remains ineligible until its next period begins. At t=0, all tasks become eligible and have deadlines $d_1 = 1$, $d_2 = d_3 = 2$. Since tasks are picked in EDF order, tasks 1 and 2 get to run on the two processors (assuming that the tie between

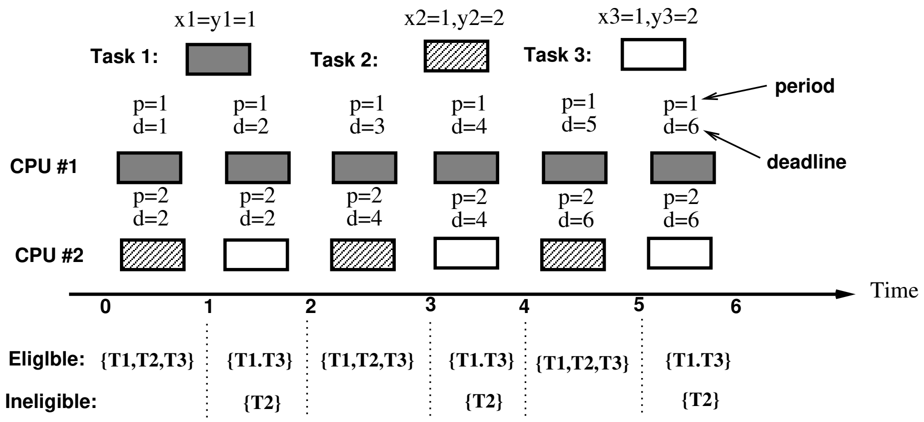


Figure 1: Use of deadlines and periods to achieve proportionate allocation.

tasks 2 and 3 is resolved in favor of task 2). Task 2 then becomes ineligible until $t = 2$ (the start of its next period). Task 1 becomes eligible again since its period is 1, while task 3 is already eligible. Since there are only two eligible tasks, tasks 1 and 3 run next. The whole process repeats from this point on. Figure 1 illustrates this scenario.

To intuitively understand how the eligibility criteria and deadlines are determined, let us assume that the quantum length=1, that each task always runs for an entire quantum, and that there are no arrivals or departures of tasks into the system. The actual scheduling algorithm does not make any of these assumptions; we do so here for simplicity of exposition. Let $m_i(t)$ be the number of times that task i has been run up to time t, where time 0 is the instant in time before the first quantum, time 1 is the instant in time between the first and second quanta, and so on. With these assumptions, to maintain P-fairness, we require that for all times t and tasks i,

$$\left\lfloor \left(\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \right) \cdot t \cdot p \right\rfloor \le m_i(t) \le \left\lceil \left(\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \right) \cdot t \cdot p \right\rceil .$$

where $(t \cdot p)$ is the total processing capacity on the p processors in time $[0, t)$. The eligibility requirements ensure that $m_i(t)$ never exceeds this range, and the deadlines ensure that $m_i(t)$ never falls short of this range. In particular, for task i to be run during a quantum, it must be the case that at the end of that quantum, $m_i(t)$ is not too large. Thus, we specify that task i is eligible to be run at time t only if

$$m_i(t) + 1 \le \left\lceil \left(\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \right) \cdot (t+1) \cdot p \right\rceil . \tag{2}$$

The deadlines ensure that a job is always run early enough that $m_i(t)$ never becomes too small. Thus, at time t we specify the deadline for the completion of the next run of task i (which will be the $(m_i(t) + 1)$th run) to be the first time t' such that

$$m_i(t) + 1 \le \left\lfloor \left(\frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \right) \cdot t' \cdot p \right\rfloor .$$

Since $m_i(t)$ and t' are always integers, this is equivalent to setting

$$t' = \left\lceil (m_i(t) + 1) \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p \cdot \phi_i} \right) \right\rceil. \tag{3}$$

With our assumptions (no arrivals or departures, and every task always runs for a full quantum), it can be shown that, if at every time step, we run the p eligible tasks with smallest deadlines (with suitable rules for breaking ties, described later in Section 3.3), then no task will ever miss its deadline. This, combined with the eligibility requirements, ensures that the resulting schedule of tasks is P-fair. That schedule is also work-conserving.

Since the actual scenario where we apply this algorithm has both variable length quantum durations, as well as arrivals and departures, the actual DFS algorithm uses a slightly different method for accounting for the amount of CPU service that each task has achieved. This greatly simplifies the accounting for the scenario we need to deal with. We also see in Section 4 that in this more difficult scenario, the algorithm is not work-conserving, and we shall remedy this by enhancing the DFS algorithm with an auxiliary work-conserving algorithm. As we shall see, the method of accounting that we use for the DFS algorithm also interfaces very easily with these enhancements.

We now describe the accounting method employed by DFS. Let S_i denote the weighted CPU service received by a task so far. In GPS-based algorithms such as WFQ [9] and SFQ [13], the quantity S_i is referred to as the *start tag* of task i; we use the same terminology here. All tasks that are initially in the system start with a value of S_i set to 0. Whenever task i is run, S_i is incremented as $S_i = S_i + \frac{1}{\phi_i}$, so that after running $m_i(t)$ times, the start tag of task i would be $S_i = \frac{m_i(t)}{\phi_i}$. Next, we define the *virtual time* v in the system as the weighted average of the progress made by all the tasks in the system at time t:

$$v = \frac{\sum_j \phi_j \cdot S_j}{\sum_j \phi_j}.$$

Note that $(\sum_j \phi_j \cdot S_j)$ is the total CPU service used by all the tasks in the system, so that at time t, this quantity would be equal to $(t \cdot p)$. Thus, substituting $S_i = \frac{m_i(t)}{\phi_i}$ and $v = \frac{t \cdot p}{\sum_j \phi_j}$ into (2), we see that the eligibility criterion becomes

$$S_i \cdot \phi_i + 1 \leq \left\lceil \phi_i \cdot \left(v + \frac{p}{\sum_j \phi_j} \right) \right\rceil.$$

Finally, we define F_i, the *finish tag* of task i, to be the weighted CPU service received by task i at the end of its next run. Then, $F_i = S_i + \frac{1}{\phi_i}$. Hence, substituting $F_i = \frac{m_i(t)+1}{\phi_i}$ into (3), we see that the deadline for task i becomes

$$t' = \left\lceil \left(\frac{\sum_j \phi_j}{p} \right) \cdot F_i \right\rceil.$$

Together, the eligibility condition and the deadlines enable DFS to ensure P-fair allocation. Having provided the intuition for our algorithm, in what follows, we provide the details of our scheduling algorithm.

3.3 Details of the Scheduling Algorithm

The precise DFS algorithm is as follows:

- Each task in the system is associated with a share ϕ_i, a start tag S_i and a finish tag F_i. When a new task arrives, its start tag is initialized as $S_i = v$, where v is the current virtual time of the system (defined below). When a task runs on a processor, its start tag is updated at the end of the quantum as $S_i = S_i + \frac{q}{\phi_i}$, where q is the duration for which the thread ran in that quantum. If a blocked task wakes up, its start tag is set to the maximum of its previous start tag and the virtual time. Thus, we have

$$S_i = \begin{cases} \max(S_i, v) & \text{if the thread just woke up} \\ S_i + \frac{q}{\phi_i} & \text{if the thread is run on a processor} \end{cases} \tag{4}$$

  After computing the start tag, the new finish tag of the task is computed as $F_i = S_i + \frac{\bar{q}}{\phi_i}$, where $\bar{q}$ is the maximum amount of time that task i can run the next time it is scheduled. Note that, if task i blocked during the last quantum it was run, it will only be run for some fraction of a quantum the next time it is scheduled, and so $\bar{q}$ may be smaller than q_{max}.

- Initially the virtual time of the system is zero. At any instant, the virtual time is defined to be the weighted average of the CPU service received by all currently runnable tasks. Defined as such, the virtual time may not monotonically increase if a runnable task with a start tag that is above average departs. To ensure monotonicity, we set v to the maximum of its previous value and the current average CPU service. That is,

$$v = \max\left(v, \frac{\sum_{j=1}^{n} \phi_j \cdot S_j}{\sum_{j=1}^{n} \phi_j}\right) \tag{5}$$

  If all processors are idle, the virtual time remains unchanged and is set to the start tag (on departure) of the thread that ran last.

- At each scheduling instance, DFS computes the set of eligible tasks from the set of all runnable tasks and then computes their deadlines as follows, where q_{max} is the maximum size of a quantum.

  - *Eligibility Criterion:* A task is eligible if it satisfies the following condition.

$$\frac{S_i \phi_i}{q_{max}} + 1 \leq \left\lceil \phi_i \left(\frac{v}{q_{max}} + \frac{p}{\sum_{j=1}^{n} \phi_j} \right) \right\rceil . \tag{6}$$

  - *Deadline:* Each eligible task is stamped with a deadline of

$$\left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil \tag{7}$$

DFS then picks the eligible task with the smallest deadline and schedules it for execution. Ties are broken using the following two tie-breaking rules:

- Rule 1: If two (or more) eligible tasks have the same deadline, pick the task i (if one exists) such that

$$\left\lfloor \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rfloor < \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil .$$

  Intuitively, such a task becomes eligible for its next period before its current deadline expires, and hence, we can have more eligible tasks in the system if this task is given preference to one that becomes eligible later than its deadline.

- Rule 2: If multiple tasks satisfy rule 1, then pick the task with the maximum value of $\lceil G_i \rceil$, where, G_i is the *group deadline* [1] of the task i, and is computed as follows.

$$G_i = 0 \quad \text{if} \left(\frac{p \cdot \phi_i}{\sum_{j=1}^{n} \phi_j} \right) < \frac{1}{2}.$$

  Otherwise, initially,

$$G_i = \frac{p \cdot \phi_i}{(\sum_{j=1}^{n} \phi_j) - p \cdot \phi_i}.$$

  From then on, whenever

$$\lceil G_i \rceil \leq \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil ,$$

  G_i is incremented by $\frac{\sum_{j=1}^{n} \phi_j}{(\sum_{j=1}^{n} \phi_j) - p \cdot \phi_i}$.

  Intuitively, this is the task that has the most severe constraints on its subsequent deadlines.

Any further ties are broken arbitrarily. These tie-breaking rules are required to ensure P-fairness in the ideal scenario where there are no arrivals or departures, and every task always runs for a full quantum.

3.4 Properties of DFS

We now show that DFS is *P-fair* as well as *work-conserving* in an ideal system model that makes the following assumptions. We assume that the ideal system model is a p-CPU symmetric multiprocessor system running a fixed set of n tasks. Further, we assume that the quanta of all the CPUs are synchronized. This means that (i) quantum lengths are fixed (without loss of generality, assume quantum length to be 1), and (ii) each time the scheduler is called, it picks a set of p tasks to run on the p CPUs for the next quantum duration. Finally, we assume that there is no processor affinity, i.e., any task can be executed on any CPU.

Given such a model, we show that DFS reduces to a P-fair scheduling algorithm proposed in [1], and using this reduction, we prove the properties achieved by DFS in such an ideal model. In the following, we define a *feasible set* of tasks to be one in which each task i with share ϕ_i satisfies the *weight feasibility constraint* (Equation 1).

Theorem 1 *Given a set of feasible tasks, DFS always generates a P-fair schedule.*

Proof:

To prove this theorem, we show that, in the ideal system model, DFS reduces to a P-fair scheduling algorithm proposed in [1] (we would refer to this as the *PF-priority* algorithm). Thus, we show that the schedule produced by DFS is exactly the same as that produced by the PF-priority algorithm, which has been proved to be a P-fair schedule in [1]. For our proof, we distinguish between a *time quantum (slot)* which we define as the execution unit for a single CPU, and *time unit* which we define as the elapsed time measured in quantum units. This implies that the system as a whole executes p quanta every time unit.

To reduce DFS to PF-priority in the ideal system model, we show the equivalence of the concepts used in the two algorithms. These concept equivalences are outlined below:

- *Periodic Tasks:*

  In PF-priority, each task T is assumed to be periodic with a requirement $(T.e, T.p)$, where $T.e$ is the task's *execution cost* and $T.p$ is the task's *period*. This means that the task T has to execute for $T.e$ time quanta (slots) every $T.p$ time units.

  We will refer to a generic task as i, and refer to its execution cost and period as x_i and y_i respectively. Thus, for the PF-priority algorithm,

$$\frac{x_i}{y_i} = \frac{T.e}{T.p}. \tag{8}$$

  In the case of DFS, each task i is assumed to have a weight ϕ_i, and the CPU share it receives is $\frac{\phi_i}{\sum_{j=1}^{n} \phi_j}$. Note that, if the task is considered periodic with a requirement (x_i, y_i), then, it executes x_i time quanta every $(p \cdot y_i)$ time quanta (as the number of time quanta executed every time unit in the system is p). Hence,

$$\frac{x_i}{p \cdot y_i} = \frac{\phi_i}{\sum_{j=1}^{n} \phi_j}$$

  Thus, for DFS,

$$\frac{x_i}{y_i} = p \cdot \frac{\phi_i}{\sum_{j=1}^{n} \phi_j}. \tag{9}$$

- *Subtasks and runs:*

  In the case of PF-priority, each task T is further subdivided into subtasks, each of which needs to execute for one quantum. The k^{th} subtask is referred to as T_k.

  Equivalently, in case of DFS, each task i consists of a series of runs of one quantum each. The number of runs completed by the task at time t is denoted by $m_i(t)$.

- *Release times and eligibility criteria:*

  In the case of PF-priority, each subtask is released at a specific time into the system, called its *pseudo-release time*. The k^{th} subtask T_k is released at time $r(T_k)$ such that

$$r(T_k) = \left\lfloor \frac{(k-1) \cdot T.p}{T.e} \right\rfloor$$

Using (8), we have

$$r(T_k) = \left\lfloor (k-1) \cdot \frac{y_i}{x_i} \right\rfloor \tag{10}$$

In the case of DFS, each task i has to satisfy an *eligibility criterion* to be eligible to run. This eligibility criterion is defined in (6) as

$$\frac{S_i \phi_i}{q_{max}} + 1 \leq \left\lceil \phi_i \left(\frac{v}{q_{max}} + \frac{p}{\sum_{j=1}^{n} \phi_j} \right) \right\rceil$$

Since quantum size is assumed to be fixed (and equal to 1), using the definitions of S_i and v as defined in section 3.2 (namely, $S_i = \dfrac{m_i(t)}{\phi_i}$ and $v = \dfrac{t \cdot p}{\sum_{j=1}^{n} \phi_j}$) and (9), the eligibility criterion for the task at time t becomes

$$m_i(t) + 1 \leq \left\lceil (t+1) \cdot \frac{x_i}{y_i} \right\rceil$$

Thus, a task becomes eligible (is released) for its k^{th} run at the minimum time $t = r_k$ which satisfies the above condition. Note that $k = m_i(t) + 1$ in this case. This is equivalent to saying that time r_k satisfies

$$k = \left\lceil (r_k + 1) \cdot \frac{x_i}{y_i} \right\rceil$$

and

$$k = \left\lceil r_k \cdot \frac{x_i}{y_i} \right\rceil + 1$$

Using the definition of the ceiling function, these equations can be rewritten as

$$(r_k + 1) \cdot \frac{x_i}{y_i} \leq k < (r_k + 1) \cdot \frac{x_i}{y_i} + 1$$

and

$$r_k \cdot \frac{x_i}{y_i} \leq k - 1 < r_k \cdot \frac{x_i}{y_i} + 1$$

Combining these two sets of inequalities, we get

$$(k-1) \cdot \frac{y_i}{x_i} - 1 < r_k \leq (k-1) \cdot \frac{y_i}{x_i}$$

Using the definition of the floor function, this is equivalent to

$$r_k = \left\lfloor (k-1) \cdot \frac{y_i}{x_i} \right\rfloor \tag{11}$$

This is the same as (10) which implies that both DFS and PF-priority use the same release times for their subtasks (runs).

- *Deadlines:*

  In the case of PF-priority, each subtask is required to *start* execution by a specific time called its *pseudo-deadline*. The pseudo-deadline of the k^{th} subtask T_k is defined as the time $d(T_k)$ such that

$$d(T_k) = \left\lceil \frac{k \cdot T.p}{T.e} \right\rceil - 1$$

  Using (8), we have

$$d(T_k) = \left\lceil k \cdot \frac{y_i}{x_i} \right\rceil - 1 \qquad (12)$$

  In the case of DFS, each task is required to *finish* execution by a specific time called its *deadline*. Thus, at time t, the deadline for the task's next run is defined in (7) as

$$d(t) = \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil$$

  Again, since quantum size is assumed to be fixed (and equal to 1), using the definition of F_i as defined in section 3.2 (namely, $F_i = \dfrac{(m_i(t)+1)}{\phi_i} = \dfrac{k}{\phi_i}$, assuming the next run is the task's k^{th} run) and (9), we have the deadline for the k^{th} run as

$$d_k = \left\lceil k \cdot \frac{y_i}{x_i} \right\rceil \qquad (13)$$

  Comparing (12) and (13), we can see that both DFS and PF-priority assign the same deadlines to their subtasks (runs).[3]

  Both DFS and PF-priority schedule the eligible (or released) tasks (subtasks) in the order of their deadlines (pseudo-deadlines). If two tasks have the same deadlines, then they apply the following tie-breaking rules.

- *Tie-breaking Rule 1:*

  If two released subtasks have the same deadline, then PF-priority gives precedence to the subtask T_k (if one exists) for which

$$r(T_{k+1}) = d(T_k). \qquad (14)$$

  DFS uses the following tie-breaking rule (section 3.3) to decide between eligible tasks with the same deadline. It gives precedence to the task i (if one exists) such that

$$\left\lfloor \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rfloor < \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil .$$

[3]Note that the difference of 1 in the deadline values is due to the way they are defined in each algorithm, with DFS defining the deadline as the *end* of the last possible quantum and PF-priority defining the deadline as the *start* of the last possible quantum. Further, this difference does not affect the schedules of the two algorithms, as the tasks are chosen in *order* of their deadlines, which is not affected by this difference of 1 quantum.

Again, using the definition of F_i, etc., this rule can be written as

$$\left\lfloor k \cdot \frac{y_i}{x_i} \right\rfloor < \left\lceil k \cdot \frac{y_i}{x_i} \right\rceil,$$

which is the same as

$$r_{k+1} = d_k - 1, \tag{15}$$

using the definitions of r_{k+1} and d_k, and the properties of the floor and ceiling functions.

From (14) and (15), we can see that both DFS and PF-priority have the same tie-breaking rule 1 (Recall the difference in the definition of deadlines for the two algorithms).

- *Tie-breaking Rule 2:*

  PF-priority defines the notion of a *group deadline* for a subtask of a task T. If two subtasks are tied even after applying the tie-breaking rule 1, then PF-priority gives precedence to the subtask with the *higher* value of its group deadline.

  The group deadline $G(T_k)$ for the k^{th} subtask of a task is defined as follows.

  First of all, define a job J to consist of all the subtasks in a period $T.p$.
  If $\frac{T.e}{T.p} < \frac{1}{2}$, then, $G(T_k) = 0$ (Such a task is called a *light* task).
  Otherwise (for a *heavy* task), the j^{th} group deadline in a job J is computed as

$$t_j = \left\lceil \frac{T.e + (j-1) \cdot T.p}{T.p - T.e} \right\rceil$$

  Thus, if $k = l \cdot T.e + k'$, then, $G(T_k)$ is defined to be smallest $t = l \cdot T.p + t_j$ such that $t > d(T_k)$.

  Thus, the group deadlines form the sequence

$$l \cdot y_i + \left\lceil \frac{x_i + (j-1) \cdot y_i}{y_i - x_i} \right\rceil, \forall l \geq 0, 0 \leq j \leq y_i - x_i, \tag{16}$$

  using the definition of x_i and y_i from (8).

  DFS borrows the definition of group deadlines from the PF-priorities algorithm. It defines the notion of group deadline G_i of a task i as follows.

  Again, just like PF-priorities, if $\frac{p \cdot \phi_i}{\sum_{j=1}^{n} \phi_j} < \frac{1}{2}$, i.e., if $\frac{x_i}{y_i} < \frac{1}{2}$, then, $G_i = 0$.
  Otherwise, G_i is computed as follows. Initially,

$$\begin{aligned} G_i &= \frac{p \cdot \phi_i}{(\sum_{j=1}^{n} \phi_j) - p \cdot \phi_i} \\ &= \frac{x_i}{y_i - x_i} \end{aligned}$$

  From then on, G_i is incremented by

$$\frac{\sum_{j=1}^{n} \phi_j}{(\sum_{j=1}^{n} \phi_j) - p \cdot \phi_i} = \frac{y_i}{y_i - x_i}$$

whenever

$$\begin{aligned} \lceil G_i \rceil &\leq \left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil \\ \text{i.e.,} \quad \lceil G_i \rceil &\leq d_k, \end{aligned}$$

where, d_k is the pseudo-deadline for the k^{th} run of task i.

DFS gives precedence to a task with higher value of $\lceil G_i \rceil$. It follows that the value of $\lceil G_i \rceil$ at any time t is the smallest value $> d_k$ from the sequence

$$l \cdot y_i + \left\lceil \frac{x_i + (j-1) \cdot y_i}{y_i - x_i} \right\rceil, \forall l \geq 0, 0 \leq j \leq y_i - x_i \tag{17}$$

From the sequences 16 and 17, it follows that both DFS and PF-priority use the same tie-breaking rule 2 as well.

Any further ties are broken arbitrarily by both DFS and PF-priority.

From the equivalence relations shown between the concepts used by DFS and PF-priority above and their rules for selecting tasks, it follows that at each time instant, DFS and PF-priority make identical choices for the next set of tasks to run. Thus, they produce identical schedules. Since it has been proven in [1] that PF-priority produces a P-fair schedule, we have shown that DFS produces a P-fair schedule as well.

■

Theorem 2 *Given a set of feasible tasks, DFS is work-conserving.*

Proof:

Using the relation given in (9), the weight feasibility constraint (1) can be rewritten as

$$\frac{x_i}{y_i} \leq 1$$

This condition specifies that no task asks to run more than once every time unit or on more than one CPU simultaneously.

Further, again using (9), we have,

$$\sum_{j=1}^{n} \frac{x_i}{y_i} = p \cdot \sum_{j=1}^{n} \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} = p$$

This implies that if every task i executes x_i quanta every y_i time units, then the total number of time quanta used up by all the tasks during every period of $\sum_{j=1}^{n} y_i$ time units is exactly $p \cdot \sum_{j=1}^{n} y_i$. In other words, the utilization of each CPU in every period is exactly 1.

By Theorem 1, DFS produces a P-fair schedule. By the definition of P-fairness [6], every P-fair schedule is also a periodic schedule. Thus, DFS ensures that every task i executes x_i quanta every y_i time units. Hence, the CPU utilization for a DFS schedule is 1, i.e., no CPU is idle as long as there are runnable tasks in the system.

This proves that DFS is work-conserving for a feasible set of tasks.

■

In the next two sections, we examine two practical issues, namely work-conserving behavior and processor affinities, that arise when implementing DFS in a multiprocessor operating system.

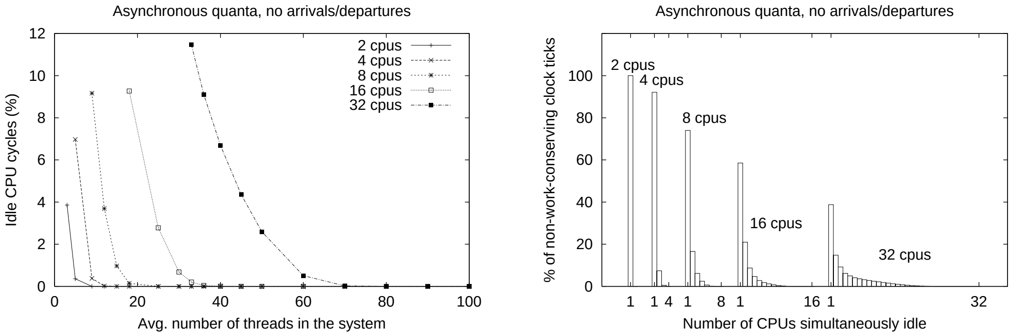
4 Ensuring Work-conserving Behavior in DFS

As indicated in Section 3.4, DFS is provably work-conserving under the ideal system model assumptions of a fixed task set and synchronized fixed length quanta. However, neither assumption holds in a typical multiprocessor system. In this section, we examine via a simulation study if DFS is work-conserving in the absence of these assumptions. It is possible for DFS to become non-work-conserving since the scheduler might mark certain runnable tasks as ineligible, resulting in fewer eligible tasks than processors (causing one or more processors to idle even in the presence of runnable tasks in the system). In what follows, we first present the methodology employed for our simulations and then present our results.

4.1 Behavior of DFS in a Conventional Operating System Environment

The methodology for our simulation study is as follows. We start with an ideal system model that assumes a fixed task set and synchronized and fixed length quanta. We then relax each assumption in turn and study the impact of doing so on the work-conserving nature of the scheduler. Specifically, we start with an ideal system where the set of tasks is static, quanta are fixed and synchronized, and tasks are scheduled on the p processors simultaneously. Next we add asynchrony to this system by allowing each processor to independently invoke the scheduler when its current quantum ends (i.e., tasks are scheduled one at a time instead of p at a time; the quantum duration and the task set remain fixed). We then allow variable quantum lengths in this system by letting the quantum duration vary on different processors. Finally, we allow arrivals and departures of tasks in the system so as to allow the task set to vary over time. At each step, we measure the percentage of CPU cycles for which the system becomes non-work-conserving and the number of processors that are simultaneously idle in the non-work-conserving mode. Such a step-by-step study helps us to determine if the system exhibits non-work-conserving behavior, and if so, the primary cause for this behavior. If our simulations indicate that the percentage of time for which the system is non-work-conserving is zero or small, then a P-fair scheduler such as DFS can be instantiated in a conventional multiprocessor operating system without any modifications. In contrast, if the system becomes non-work-conserving for significant durations, then we will need to consider remedies to correct this behavior.

To conduct our simulation study, we simulate multiprocessor systems with 2, 4, 8, 16 and 32 processors. We initialize the system with a certain number of tasks. In the scenario where arrivals and departures are allowed, we generate these events using exponential distributions for inter-arrival and inter-departure times; the mean rates of arrivals and departures are chosen to be identical to keep the system stable. The processor share ϕ_i requested by each task is chosen randomly from a uniform distribution and we ensure that requested shares are feasible at all times. Similar to most operating systems, our simulations measure time in units of clock ticks (for instance, Linux measures its quanta in units of jiffies, each jiffy being equal to 10ms). The maximum quantum duration is set to 10 ticks. In the scenario where the quantum duration can vary, we do so by using a uniform distribution from 1 to 10 ticks. We simulate each of our four scenarios for 10,000 ticks and repeat the simulation 1,000 times, each with a different seed (so as to simulate a wide range of task mixes). We obtain the following results from our study:



(a) Variation with number of tasks (b) Number of simultaneously idle processors

Figure 2: Effect of asynchronous quanta on the work-conserving behavior.

- *Ideal system:* As expected, our simulation results (not shown here) showed that DFS is work-conserving in an ideal system where the set of tasks is fixed and quanta are synchronized and of fixed length, which conforms to the theoretical properties (Theorem 2) proved in Section 3.4.

- *Asynchronous quanta:* We add asynchrony to the system by allowing each processor to independently invoke the scheduler when its current quantum ends; the length of each quantum is fixed and so are the number of tasks in the system. What this means is that instead of scheduling p tasks on the p CPUs simultaneously, each CPU schedules a task individually as happens on real multiprocessor systems.

  As shown in Figure 2(a), this causes the system to become non-work-conserving. The non-work-conserving behavior is most pronounced when the number of tasks in the system is close to the number of processors; for such cases, the fraction of CPU cycles wasted due to one or more processors being idle is as large as 12%. The figure also shows that increasing the number of runnable tasks causes an increase in the number of eligible tasks in the system and thereby reduces the chances of the system becoming non-work-conserving. Figure 2(b) plots a histogram of the number of processors that simultaneously remain idle when the system is non-work-conserving. As shown in the figure, multiple processors can simultaneously become idle in the non-work-conserving state, which degrades overall system utilization.

  The reason for this non-work-conserving in the presence of asynchronous quanta is the asynchronous updates made by the algorithm for each CPU. The scheduling algorithm updates the various quantities such as the start tag and finish tag of the running task and the virtual time whenever a CPU is scheduled. These updates happen in an asynchronous manner, so that the scheduler does not have a completely up-to-date state of the system at all times. This discrepancy leads to some tasks being considered ineligible even when they would actually be eligible for running. This causes some CPUs to remain idle even in the presence of runnable tasks.

- *Variable length quanta:* Next, we let the quantum lengths vary but keep the number of tasks in the system fixed. The results obtained for this scenario (asynchronous variable-length quanta) are similar to those obtained in the previous scenario (asynchronous fixed-length quanta), and hence, are not shown here. These results indicate that the asynchrony in scheduling is the primary cause for non-work-conserving behavior and variable length quanta does not substantially worsen this behavior.

- *Arrivals and departures:* Our final scenario adds arrivals and departures to the system. Here, we use a Poisson process to introduce task arrivals into the system and another Poisson process to introduce task departures. These arrival and departure processes emulate the task blocking/non-blocking events (such as page faults and I/O events) that take place in a real system. While an arrival event adds a task to the system run queue, a departure event removes a currently running task from the run queue. We choose a running task to depart at a departure event, because most blocking (departure) events typically affect only tasks that are currently running in a real system. In our simulation, the departing task is chosen at random from among the currently running tasks. This ensures that the expected runtime of any task is independent of its share, as a task is equally likely to depart every time it is run.

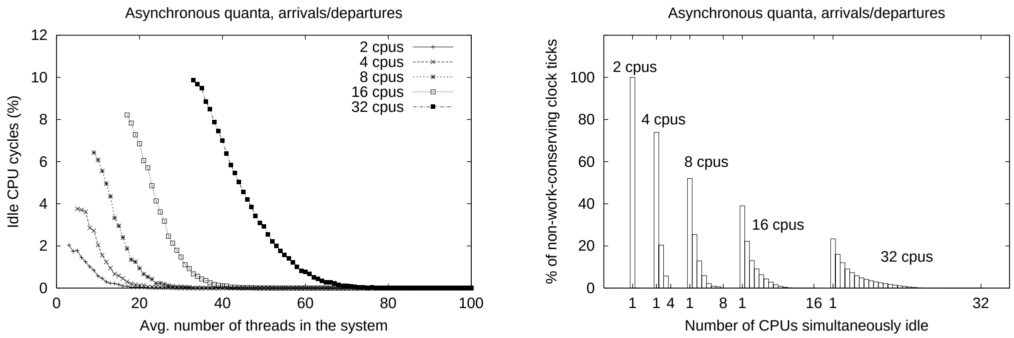
  Our results again show that the system becomes non-work-conserving especially when the number of tasks is close to the number of processors (see Figure 3). Interestingly, we find that the average fraction of CPU cycles that are wasted *decreases* slightly as compared to the previous two scenarios (observe this by comparing Figures 3(a) and 2(a)). This decrease is caused by new arrivals, each of which introduces an additional eligible task into the system, causing an idle processor (if one exists) to schedule this task. Without such arrivals, the processor would have idled until an existing ineligible task became eligible. Departures, which should have the opposite effect, seem to have a smaller impact on the non-work-conserving behavior. This diminished effect is because a task departs while it is running (as explained above)—thus, it does not add to the number of ineligible tasks in the system on finishing, that might have adversely affected the work-conserving behavior.

  Finally, Figure 4 plots the effect of varying the arrival/departure rate on the system behavior. The figure, plotted on a log scale, shows that increasing the inter-arrival times causes a slow increase in the fraction of the time the system is non-work-conserving (since a larger inter-arrival time implies fewer arrivals, which then reduces the probability that an idle processor schedules a newly arriving task).

We conclude from our simulation study that DFS can exhibit non-work-conserving behavior when employed in a conventional multiprocessor operating system. Since the fraction of CPU cycles that can be wasted can be as large as 10-12%, the DFS scheduler needs to be enhanced with an additional policy that allocates idle processor bandwidth to tasks that are runnable but ineligible (so as to improve system utilization). In the rest of this section we show how to combine DFS with an auxiliary work-conserving scheduler to achieve this objective.

4.2 Combining DFS with Fair Airport Scheduling Algorithms

We draw upon the concept of *fair airport scheduling* to enhance DFS with an auxiliary policy to allocate idle bandwidth to ineligible runnable tasks. The notion of fair airport was proposed in the context of



(a) Variation with number of tasks (b) Number of simultaneously idle processors

Figure 3: Effect of arrivals and departures on the work-conserving behavior.

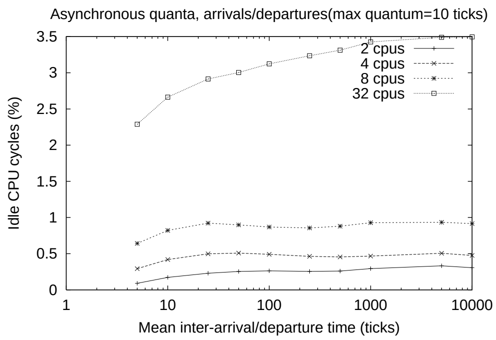


Figure 4: Effect of the arrival/departure rate on the work-conserving behavior.

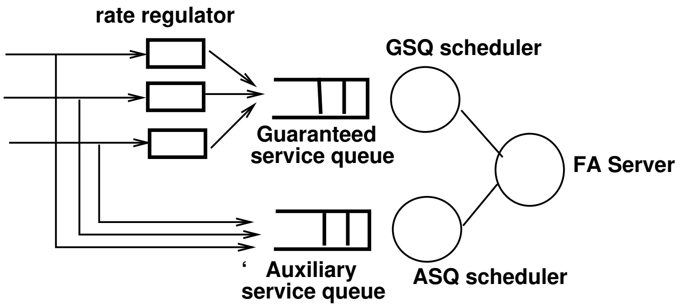


Figure 5: Fair Airport Scheduling Algorithm

scheduling packets at a network router [8, 12]. A fair airport scheduler attempts to combine a potentially non-work-conserving scheduling algorithm with an auxiliary scheduler to ensure work-conserving behavior at all times. Each packet (or task) in a fair airport scheduler joins a rate regulator and an Auxiliary Service Queue (ASQ) (see Figure 5). The rate regulator is responsible for determining when a packet is eligible to be scheduled. Once eligible, the packet passes through the regulator and joins the Guaranteed Service Queue (GSQ) and is then serviced by the GSQ scheduler. If the guaranteed service queue becomes empty, the ASQ scheduler is invoked to service packets in the ASQ (note that these are packets that are currently ineligible). The combined scheduler always gives priority to the GSQ over the ASQ—the GSQ scheduler gets to schedule packets so long as the GSQ is non-empty and the ASQ scheduler is invoked only when GSQ becomes empty. Different scheduling algorithms may be employed for servicing packets in the guaranteed service and auxiliary service queues. Depending on the exact choice of the ASQ and GSQ schedulers, it is possible to theoretically prove properties of the combined scheduling algorithm (see [8, 12] for examples).

The concept of fair airport scheduling can also be employed to schedule tasks in a multiprocessor system. Our instantiation of fair airport, referred to as DFS-FA, employs DFS as the GSQ scheduler. The rate regulator for each task is simply its eligibility criterion (Equation 6); the rate regulator then ensures that a task joins the guaranteed service queue only once in each period. The ASQ scheduler is used to service tasks if the GSQ becomes empty. By servicing tasks that are runnable but ineligible, the ASQ scheduler ensures that the combined scheduler is work-conserving at all times.

Any work-conserving scheduling algorithm can be used to instantiate the ASQ scheduler. We choose a scheduler that services ineligible tasks in the increasing order of their start tags (i.e., when the GSQ becomes empty, the task with the smallest start tag in the ASQ is scheduled for execution). There are a number of reasons for choosing this scheduling policy. The implementation of the basic DFS algorithm requires two queues—a queue for eligible tasks and one for ineligible tasks. The latter queue is sorted in order of start tags, since this is the order in which tasks become eligible and are then moved to the eligible queue (see section 6 for details). Consequently, the ASQ scheduler can be simply implemented by having the ineligible queue double up as the auxiliary service queue, and by scheduling the tasks at the head of this queue when the eligible queue (GSQ) becomes empty. Thus, our fair airport enhancement to DFS can be implemented without any additional data structures or overheads as compared to the basic DFS algorithm. Further, scheduling tasks in order of start tags is equivalent to using *Start time fair queuing* [13], a scheduling algorithm that has known fairness and delay properties. Thus, choosing this scheduling policy has the added benefit of providing provably predictable performance guarantees in the ASQ.

As a final caveat, we note that servicing ASQ tasks in order of start tags allows residual bandwidth to be allocated to tasks in proportion to their shares (i.e., enables fair redistribution of residual bandwidth). Criteria other than fairness can also be used to redistribute residual bandwidth. For instance, a priority-based scheduler can be used to service the ASQ so as to give priority to certain tasks when allocating idle bandwidth. A detailed study of such policies is beyond the scope of this paper.

5 Accounting for Processor Affinities in DFS

Another practical consideration that arises when implementing a CPU scheduler for a multiprocessor system is that of processor affinities. Each processor in a multiprocessor system employs one or more levels of memory caches. These caches store recently accessed data and instructions for each task. Scheduling a task on the same processor enables it to benefit from the data cached from the previous scheduling instances (and also eliminates the need to flush the cache on a context switch to maintain consistency). In contrast, scheduling a task on a different processor can increase the number of cache misses and degrade performance. Studies have shown that a scheduler that takes processor affinities into account while making scheduling decisions can improve the effectiveness of the cache and the overall system performance [25].

Observe that the basic DFS algorithm uses internally generated deadlines (Equation 7) to make scheduling decisions and ignores processor affinities. This limitation can be overcome by using one of two different approaches. The first approach partitions the set of tasks among the p processors such that each processor is load balanced and employs a local run queue for each processor. Each processor runs the DFS scheduler on its local run queue. Binding a task to a processor in this manner allows the processor to exploit cache locality. However, if all tasks were permanently bound to individual processors, then the load across processors would most likely be unbalanced over time (due to blocking/termination events). Consequently, periodic repartitioning of tasks among processors is necessary to maintain a balanced load. Another limitation of the approach is that P-fairness guarantees can be provided only on a per-processor basis (instead of a system-wide basis), since individual processors neither coordinate with each other nor have a balanced load.

A second approach to account for processor affinities is to employ a single global run queue and use a more sophisticated metric for making scheduling decisions. Recall that the basic DFS algorithm stamps each eligible task with a deadline (Equation 7). We modify this deadline value to incorporate processor affinities in the following manner. We define a modified *pseudo-deadline* D for an eligible task as a function of its DFS-deadline and its affinity for the processor currently being scheduled:

$$D = d + \alpha \cdot \mathcal{A}, \tag{18}$$

where d denotes the DFS-deadline of the task, α is a positive constant and $\mathcal{A}$ is a measure of its affinity for the processor being scheduled. For instance, in the simplest case, $\mathcal{A}$ is defined as 0 for the processor that a task ran on last and 1 for all other processors. Thus, $\alpha \cdot \mathcal{A}$ represents the penalty for scheduling a task with poor processor affinity. The scheduler then picks the task with the minimum pseudo-deadline.

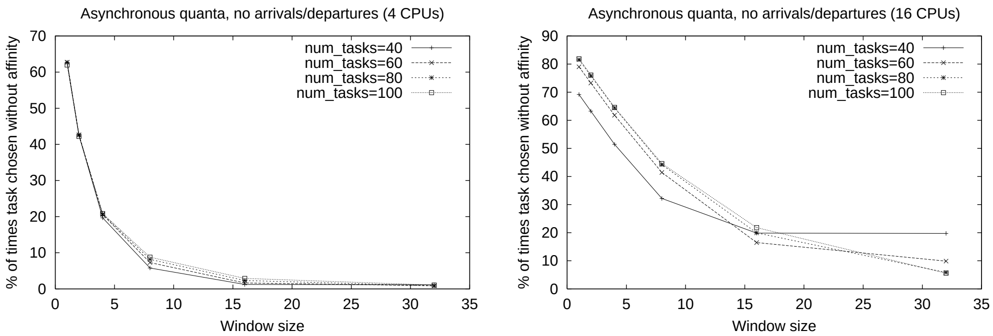
Assuming that the DFS algorithm maintains a list of eligible tasks sorted on their deadlines, the scheduling algorithm would then need to compute the pseudo-deadline D of each task in this list before picking the task with the minimum value of D (since D is a processor dependent metric, it is not possible to compute its value for each task a priori). This approach makes scheduling decisions linear in the number of eligible tasks, which can be expensive in systems with a large number of tasks. Scheduling decisions can be made more efficient (constant time) by defining a window $\mathcal{W}$ that limits the number of tasks that must be examined for their pseudo-deadlines before picking a task. The window represents a tradeoff between fairness guarantees and processor affinities. A small window favors fairness (by picking the tasks with short deadlines and better approximating P-fairness) but can reduce the chances of finding a task with good affinity. In the extreme case, $\mathcal{W} = 1$ reduces the scheduler to a pure DFS scheduler. In contrast, a large window can increases the chances of finding a task with an affinity for the processor but can increase unfairness. Thus, $\mathcal{W}$ is a tunable parameter that allows us to balance three conflicting tradeoffs—fairness, scheduling efficiency, and processor affinities.

We conducted simulation experiments to determine the effectiveness of using pseudo-deadlines to account for processor affinities. We explored the parameter space by varying the number of processors from 2 to 32, the number of tasks from 1 to 100, and the window size from 1 to 32. For each combination of these parameters, we computed the percentage of times the scheduler is successfully able to pick a task with an affinity for the processor and also the resulting unfairness in the allocation. Figure 6 shows our results for some combinations of these parameters (we omit other results due to space constraints). The figure shows that increasing $\mathcal{W}$ improves the effectiveness of the algorithm in picking a task with processor affinity (examining a larger number of tasks increases the chances of picking the "right" task). As a rule of thumb, we recommend that the window size be set to the number of processors ($\mathcal{W} = p$) to balance the tradeoffs of scheduling efficiency and processor affinity. As shown in Table 1, using this rule of thumb does not greatly increase unfairness— tasks remain within one quantum of their due share for 83% of the time and within two quanta away from their P-fair share for 99% of the time. As an aside, since we simulate a system with asynchronous variable length quanta, even when $\mathcal{W} = 1$, the basic DFS algorithm shows some deviation from strict P-fairness (which would have constrained all tasks within one quantum of their ideal share). These results indicate that using pseudo-deadlines can be an effective technique to handle processor affinities in small to medium-sized multiprocessor systems (< 32 processors).

In what follows we discuss the implementation of DFS in the Linux kernel.

6 Implementation Details

We have implemented the DFS algorithm as well as the two enhancements discussed in Sections 4 and 5 into the Linux kernel (source code for our implementation is available from our web site). Our DFS scheduler, implemented in version 2.2.14 of the kernel, replaces the standard time-sharing scheduler in Linux. Our implementation allows each task to specify a share ϕ_i. Tasks can dynamically change or query their shares using two new system calls, `setshare` and `getshare`. These system calls are described in Table 2.



(a) 4-CPU system

(b) 16-CPU system

Figure 6: Effect of Window size on Processor Affinity

Table 1: Deviation from P-fairness for a 4-processor system

Window Size	Deviation from ideal share (% of scheduling instances)		
	0-1 quanta	1-2 quanta	>2 quanta
1	83.38	16.55	0.07
2	83.44	16.50	0.06
4	83.76	16.18	0.06
8	83.99	15.95	0.06
16	83.84	16.10	0.06
32	83.51	16.42	0.07

Their interface is very similar to the Linux system calls `setpriority` and `getpriority` that are used to assign priorities to tasks in the standard time-sharing scheduler.

Our implementation of DFS maintains two run queues—one for eligible tasks and the other for ineligible tasks (see Figure 7). The former queue consists of tasks sorted in deadline order; DFS services these tasks using EDF. The latter queue consists of tasks sorted on their start tags, since this is the order in which tasks become eligible. Once eligible, a task is removed from the ineligible queue and inserted into the eligible queue.

The actual scheduler works as follows. Whenever a task's quantum expires or it blocks for I/O or departs, the Linux kernel invokes the DFS scheduler. The scheduler first updates the start tag and finish tag of the

Table 2: System calls used for controlling weights of tasks

Syscall	Description
`int setshare(int which, int who, int share)`	Set the share of a process, process group or user
`int getshare(int which, int share)`	Return processor share of a process, process group or user

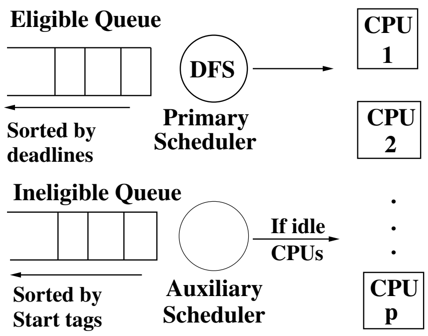


Figure 7: DFS-FA Scheduler

task relinquishing the CPU. Next, it recomputes the virtual time based on the start tags of all the runnable tasks. Based on this virtual time, it determines if any ineligible tasks have become eligible, and if so, moves them from the ineligible queue to the eligible queue in deadline order. If the task relinquishing the CPU is still eligible, it is reinserted into the eligible queue, else it is marked ineligible and inserted into the ineligible queue in order of start tags. The scheduler then picks the task at the head of the eligible queue and schedules it for execution.

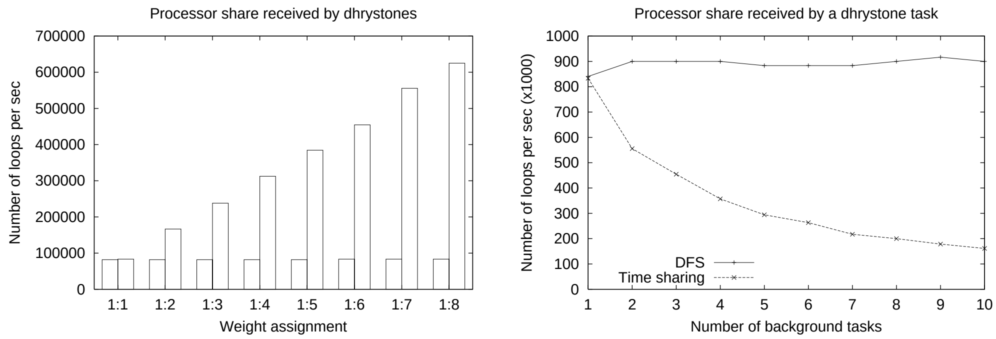
The two enhancements proposed to the DFS algorithm are implemented as follows:

- *Fair airport:* The fair airport enhancement can be implemented by simply using the eligible queue as the GSQ and the ineligible queue as the ASQ. If the eligible queue becomes empty, the scheduler picks the task at the head of the ineligible queue and schedules it for execution. Thus, the enhancement can be implemented with no additional overheads and results in work-conserving behavior.

- *Processor affinities:* We consider the approach that employs a single global run queue and pseudo-deadlines to account for processor affinities (and do not consider the approach that employs a local run queue for each processor). We assume that the window size $\mathcal{W}$ is specified at boot time. At each scheduling instance, the DFS scheduler computes the pseudo-deadlines of the first $\mathcal{W}$ tasks in the eligible queue and schedules the task with the minimum pseudo-deadline value (see (18)). By choosing an appropriate value of α in (18), the scheduler can be biased appropriately towards picking tasks with processor affinities (larger values of α increase the bias towards tasks with an affinity for a processor).

7 Experimental Evaluation

In this section, we describe the results of our experimental evaluation. We conducted experiments to (i) demonstrate proportional allocation property of DFS-FA, (ii) show the performance isolation provided by it to applications, and (iii) measure the scheduling overheads imposed by it. Where appropriate, we use the Linux time-sharing scheduler as a baseline for comparison. In what follows, we first describe our experimental test-bed, and then present the experimental results.



(a) Proportionate Allocation (b) Application Isolation

Figure 8: Proportionate Allocation and Application Isolation with DFS-FA

7.1 Experimental Setup

For our experiments, we used a 500 MHz Pentium III-based dual-processor PC with 128 MB RAM, 13GB SCSI disk and a 100 Mb/s 3-Com ethernet card (model 3c595). The PC ran the default installation of RedHat Linux 6.2. We used Linux kernel version 2.2.14 for our experiments, which employed either the time-sharing or the DFS-FA scheduler depending on the experiment. The system was lightly loaded during our experiments.

The workload for our experiments consisted of a mix of sample applications and benchmarks. These include : (i) *mpeg_play*, the Berkeley software MPEG1 decoder, (ii) *mpg123*, an audio MPEG and MP3 player, (iii) *dhrystone*, a compute-intensive benchmark for measuring integer performance, (iv) *gcc*, the GNU C compiler, (v) *RT_task*, a program that emulates a real-time task, and (vi) *lmbench*, a benchmark that measures various aspects of operating system performance. Next, we describe the results of our experimental evaluation.

7.2 Proportional Allocation and Application Isolation

We first demonstrate that DFS-FA allocates processor bandwidth to applications in proportion to their shares, and in doing so, it also isolates each of them from other misbehaving or overloaded applications. To show these properties, we conducted two experiments with a number of *dhrystone* applications. In the first experiment, we ran two dhrystone applications with relative shares of 1:1, 1:2, 1:3, 1:4, 1:5, 1:6, 1:7 and 1:8 in the presence of 20 background dhrystone applications. As can be seen from figure 8(a), the two applications receive processor bandwidth in proportion to the specified shares.

In the second experiment, we ran a dhrystone application in the presence of increasing number of background dhrystone tasks. The processor share assigned to the foreground task was always equal to the sum of the shares of the background jobs. Figure 8(b) plots the processor bandwidth received by the foreground task with increasing background load. For comparison, the same experiment was also performed with the default Linux time-sharing scheduler. As can be seen from the figure, with DFS-FA, the processor share received by the foreground application remains stable irrespective of the background load, in effect isolating

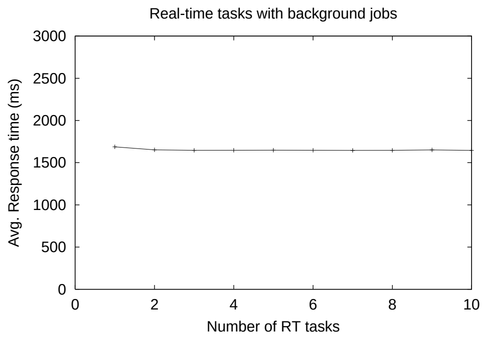


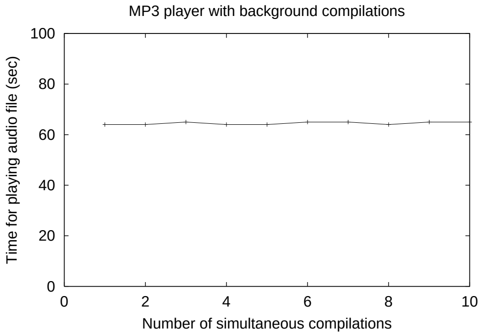
Figure 9: Performance of DFS when scheduling a mix of real-time applications.

the application from load in the system . Not surprisingly, the time-share scheduler is unable to provide such isolation. These experiments demonstrate that while DFS-FA is no longer strictly P-fair, it nevertheless achieves proportional allocation. In addition, it also manages to isolate applications from each other.

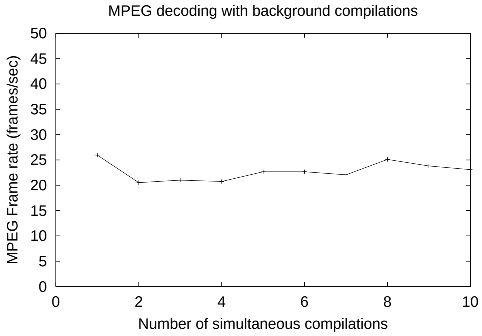
7.3 Impact on Real-Time and Multimedia Applications

In the previous subsection, we demonstrated the desirable properties of DFS-FA using a synthetic, compute-intensive benchmark. Here, we demonstrate how DFS-FA can benefit real-time and multimedia applications. To do so, we first ran an experiment with a mix of *RT_tasks*, each of which emulates a real-time task. Each task receives periodic requests and performs some computations that need to finish before the next request arrives; thus, the deadline to service each request is set to the end of the period. Each real-time task requests CPU bandwidth as (x, y) where x is the computation time per request, and y is the inter-request arrival time. In the experiment, we ran one RT_task with fixed computation and inter-arrival time, and measured its response time with increasing number of background real-time tasks. As can can be seen from figure 9, the response time is independent of the other tasks running in the system. Thus, DFS-FA can support predictable allocation for real-time tasks.

In the second experiment, we ran the streaming audio application (an MP3 player) in the presence of a large number of background compilation jobs. This scenario is typical on a desktop, where a user could be working (in this case, compiling a large application) while listening to audio music. Figure 10(a) demonstrates that the performance of the streaming audio application remains stable even in the presence of increasing background jobs. We repeated this experiment with streaming video; a software decoder was employed to decode and display a 1.5 Mb/s MPEG-1 file in the presence of other best-effort compilation jobs. Figure 10(b) shows that the frame rate of the mpeg decoder remains stable with increasing background load, but less so than the audio application. We hypothesize that the observed fluctuations in the frame rate are due to increased interference in disk accesses. The data rate of a video file is significantly larger than that of an audio file, and the increased I/O load due to the compilation jobs interfere with the reading of the MPEG-1 file from disk. Overall, these experiments demonstrate that DFS-FA can support real-time and multimedia applications.



(a) Streaming Audio



(b) Streaming Video

Figure 10: Performance of multimedia applications.

Table 3: Lmbench Results

Test	Linux	DFS
syscall overhead	0.7 μs	0.7 μs
`fork()`	400 μs	400 μs
`exec()`	2 ms	2 ms
Context switch (2 proc/ 0KB)	1 μs	5 μs
Context switch (8 proc/ 16KB)	15 μs	20 μs
Context switch (16 proc/ 64KB)	178 μs	181 μs

7.4 Scheduling Overheads

In this section, we describe the scheduling overheads imposed by the DFS-FA scheduler on the kernel. We used *lmbench*, a publicly available operating system benchmark, to measure these overheads. Lmbench was run on a lightly loaded system running the time-sharing scheduler, and again on a system running the DFS-FA algorithm. We ran the benchmark multiple times in each case to reduce experimental error. Table 3 summarizes the results we obtained. We report only those lmbench statistics that are relevant to the CPU scheduler. As can be seen from Table 3, the overhead of creating tasks (measured using `fork` and `exec` system calls) is comparable in both cases. However, the context switch overhead increases by about 3-5 μs. This overhead is insignificant compared to the quantum duration used by the Linux kernel, which is several orders of magnitude larger (typical quantum durations range from tens to hundreds of milliseconds; the default quantum duration used by the Linux kernel is 200ms).

8 Concluding Remarks

In this paper, we presented Deadline Fair Scheduling (DFS), a proportionate-fair CPU scheduling algorithm for multiprocessor servers. A particular focus of our work was to investigate practical issues in instantiating proportionate-fair schedulers in general-purpose operating systems. Our simulation results showed that characteristics of general-purpose operating systems such as the asynchrony in scheduling multiple processors, frequent arrivals and departures of tasks, and variable quantum durations can cause a P-fair scheduler such as DFS to become non-work-conserving. To overcome these limitations, we enhanced DFS using the fair airport scheduling framework to ensure work-conserving behavior at all times. We then proposed techniques to account for processor affinities while scheduling tasks in multiprocessor environments. Our resulting scheduler trades strict fairness guarantees for more practical considerations. We implemented the resulting scheduler, referred to as DFS-FA, in the Linux kernel and demonstrated its performance on real workloads. Our experimental results showed that DFS-FA can achieve proportional allocation, performance isolation and work-conserving behavior at the expense of a small increase in the scheduling overhead. We conclude that combining a proportionate-fair scheduler such as DFS with considerations such as work-conserving behavior and processor affinities is a practical approach for scheduling tasks in multiprocessor operating systems.

Acknowledgements

This research was supported in part by a NSF Career award CCR-9984030, NSF grants ANI 9977635, CDA-9502639, EIA-0080119, Intel, IBM, EMC, Sprint, and the University of Massachusetts. We would also like to thank James Anderson, Sanjoy Baruah and Krithi Ramamritham for numerous discussions on P-fair scheduling algorithms.

References

[1] J. Anderson and A. Srinivasan. A New Look at Pfair Priorities. Technical report, Dept of Computer Science, Univ. of North Carolina, 1999.

[2] J. Anderson and A. Srinivasan. Early-Release Fair Scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden*, June 2000.

[3] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[4] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the third Symposium on Operating System Design and Implementation (OSDI'99), New Orleans*, pages 45–58, February 1999.

[5] S. Baruah, J. Gehrke, and C. G. Plaxton. Fast Scheduling of Periodic Tasks on Multiple Resources. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 280–288, April 1996.

[6] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15:600–625, 1996.

[7] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, CA*, October 2000.

[8] R.L. Cruz. Service Burstiness and Dynamic Burstiness Measures: A Framework. *Journal of High Speed Networks*, 2:105–127, 1992.

[9] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, September 1989.

[10] K. Duda and D. Cheriton. Borrowed Virtual Time (BVT) Scheduling: Supporting Lantency-sensitive Threads in a General-Purpose Scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC*, pages 261–276, December 1999.

[11] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Operating System Design and Implementation (OSDI'96), Seattle*, pages 107–122, October 1996.

[12] P. Goyal and H M. Vin. Fair Airport Scheduling Algorithms. In *Proceedings of the Seventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97), St. Loius, MO*, pages 273–281, May 1997.

[13] P. Goyal, H. M. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96*, pages 157–168, August 1996.

[14] M B. Jones, D Rosu, and M Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97), Saint-Malo, France*, pages 198–211, December 1997.

[15] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.

[16] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE ICMCS'94*, May 1994.

[17] M. Moir and S Ramamurthy. Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resources. In *Proceedings of the 20th Annual IEEE Real-Time Systems Symposium, Phoenix, AZ*, December 1999.

[18] J. Nieh and M S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France*, pages 184–197, December 1997.

[19] A. Parekh and R. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks – The Single Node Case. In *Proceedings of IEEE INFOCOM '92*, pages 915–924, May 1992.

[20] A.K. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1992.

[21] A. Srinivasan and J. Anderson. Efficient Scheduling of Soft Real-time Applications on Multiprocessors. Technical report, Dept of Computer Science, Univ. of North Carolina, December 2002.

[22] A. Srinivasan and J. Anderson. Optimal Rate Based Scheduling on Multiprocessors. In *Proceedings of the ACM Symposium on Theory of Computing*, May 2002.

[23] A. Srinivasan and J. Anderson. Fair Scheduling of Dynamic Task Systems on Multiprocessors. In *Proceedings of the International Workshop on Parallel and Distributed Real-Time Systems*, April 2003.

[24] A. Srinivasan, P. Holman, and J. Anderson. Integrating Aperiodic and Recurrent Tasks on Fair-scheduled Multiprocessors. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, June 2002.

[25] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 26–40, October 1991.

[26] C. Waldspurger and W. Weihl. Stride Scheduling: Deterministic Proportional-share Resource Management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.

List of Tables

List of Figures

Affiliation of Authors

Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003

List of Footnotes

1. See http://lass.cs.umass.edu/software/gms.

2. Multimedia/streaming media applications are an important subset of the class of soft real-time applications. Note that, there could be other applications such as virtual reality that are soft real-time but do not involve streaming audio and video.

3. Note that the difference of 1 in the deadline values is due to the way they are defined in each algorithm, with DFS defining the deadline as the *end* of the last possible quantum and PF-priority defining the deadline as the *start* of the last possible quantum. Further, this difference does not affect the schedules of the two algorithms, as the tasks are chosen in *order* of their deadlines, which is not affected by this difference of 1 quantum.

Keywords

Proportionate-fairness
Multiprocessor
Scheduling
Real-time
Proportional-share
Deadline
Eligibility
Work-conserving