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

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 proportionate-fair (P-fair) schedulers into conventional operating systems. We show via a simulation study that characteristics of conventional operating systems such as the asynchrony in scheduling multiple processors, frequent arrivals and departures of tasks, and variable quantum durations can cause proportionate-fair schedulers 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 proportionate allocation, performance isolation and work-conserving behavior at the expense of a small increase in the scheduling overhead. We conclude that practical considerations such as work-conserving behavior and processor affinities when incorporated 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 [2, 3, 10, 16]. Proportionate fair schedulers are one such class of scheduling algorithms [5]. 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, 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 (e.g., multiprocessor systems).

Several P-fair schedulers have been proposed over the past few years [1, 4, 14]. Most of these research efforts have focused on theoretical analyses of these schedulers. In this paper, we consider practical issues that arise when implementing a proportionate-fair scheduler into a multiprocessor operating system kernel. Our research effort has led to several contributions. First, we propose a new P-fair scheduling algorithm referred to as *Deadline Fair Scheduling (DFS)* for multiprocessor environments. We then show using simulations that typical characteristics of multiprocessor 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. Since a non-work-conserving scheduler can cause a processor to remain idle even in the presence of runnable tasks (which reduces processor utilization), 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 considera-

*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.

tion for multiprocessor schedulers is the ability to take *processor affinities* [18] into account while making scheduling decisions—scheduling a thread 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 such as Spring [17], 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 numerous applications and benchmarks. Our results show that DFS can achieve proportionate 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 present our conclusions in Section 8.

2. Proportional-Share Scheduling and Proportionate-Fairness: Basic Concepts

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.

Several resource management mechanisms have been developed to explicitly deal with soft real-time applications [2, 10, 11, 15, 19]. 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. Schedulers based on generalized processor sharing (GPS) [15] such as weighted fair sharing [9], start-time fair queuing [11] and borrowed virtual time [10] are one class of proportional-share schedulers. West et al. [20] have described a deadline-based proportional-share scheduler which takes into account acceptable loss rates for multiple streams. Recently, several studies have focused on proportional-share scheduling in multiprocessors. Jones et al. have proposed a reservation-based scheduler for multiprocessors in [13], while relative weight-based allocation for multiprocessor systems was studied in [6].

Proportionate fair (P-fair) schedulers are another class of proportional-share schedulers. P-fairness is based on the notion of proportionate progress [5]. 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. As indicated in Section 1, 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-fairness which assumes that applications can be serviced in terms of infinitesimally small time quanta, P-fairness assumes that applications are allocated finite duration quanta (and thus is a more practical notion of fairness). However, the above definition of P-fairness assumes that the quantum duration is fixed. In practice, 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. 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 algorithms have been proposed which achieve P-fairness in an ideal model — synchronized, fixed quantum durations and a fixed task set [1, 4, 5]. In the context of multiprocessors, P-fair schedulers for static and migratable tasks have been studied in [14]. Most of these papers have focused on the theoretical aspects of P-fair scheduling. In this paper, we propose an algorithm based on the notion of P-fairness which achieves proportional-share in practical systems. This algorithm is clearly defined even when the system has variable quantum durations and arrivals and departures of tasks. Moreover, when the quantum sizes and the task set are fixed, it achieves P-fairness. To seamlessly account for non-ideal system considerations, 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

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

$\lceil \frac{\phi_i}{\sum_j \phi_j} \cdot pT \rceil$ 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).

A final dimension for classifying proportional-share 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 so long as there are runnable tasks in the system. Non-work-conserving schedulers, on the other hand, can let idle a processor 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 the processor is idle. To achieve good resource utilization, schedulers employed in conventional operating systems tend to be work-conserving in nature.

In what follows, we present a 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:

$$\frac{\phi_i}{\sum_{j=1}^{N} \phi_j} \leq \frac{1}{p} \qquad (1)$$

This condition forms the basis for admission control in our scheduler and is used to limit the number of tasks in the system. Our Deadline Fair Scheduling (DFS) algorithm achieves these allocations 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.[2] 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.

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[3]. 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,

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

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

[2] In other words, we can consider each task to consist of periodic subtasks, whose release times are determined using the eligibility criteria, and each subtask needs to execute once before its deadline expires.

[3] The actual scheduling algorithm does not make any of these assumptions; we do so here for simplicity of exposition.

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

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

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 $m_i(t)+1$st run) to be the first time t' such that

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

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

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

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, as described below), 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 lengths, as well as arrivals and departures, the actual DFS algorithm will use 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 shall also see that in this more difficult scenario, the algorithm is not work-conserving, and we shall remedy this by enhancing the basic DFS algorithm to ensure work-conserving behavior. The method of accounting that we shall use for the basic DFS algorithm also interfaces very easily with these enhancements.

To understand the accounting method, assume that for every task i, S_i denotes the CPU service received by the task so far. 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 $S_i = S_i + \frac{1}{\phi_i}$. In GPS-based algorithms such as WFQ [9] and SFQ [11], the quantity S_i is referred to as the *start tag* of task i; we use the same terminology here. Let $v = (\sum_j \phi_j \cdot S_j)/\sum_j \phi_j$. Intuitively, v is a weighted average of the progress made by the tasks in the system at time t, and is referred to as the *virtual time* in the system. Substituting $S_i = m_i(t)/\phi_i$ and $v = tp/\sum_j \phi_j$ into Equation 2, we see that the eligibility criteria becomes $S_i \cdot \phi_i + 1 \leq \lceil \phi_i(v + p/\sum_j \phi_j) \rceil$.

Let F_i, the *finish tag* of task i, be the CPU service received by task i at the end of the next quantum where task i is run. Then, $F_i = S_i + \frac{1}{\phi_i}$. Substituting $F_i = (m_i(t)+1)/\phi_i$ into Equation 3, we see that the deadline for task i becomes $t' = \left\lceil \frac{\sum_{j=1}^{n} \phi_j}{p} F_i \right\rceil$. 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 works as described below.

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

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

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 average CPU service received by a thread. That is,

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

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 threads 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.

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

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

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

DFS then picks the 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 $v \geq \lceil G_i \rceil \cdot \left(\frac{p}{\sum_{j=1}^{n} \phi_j} \right)$, 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 case where there are no arrivals or departures, and every task always runs for a full quantum.

3.4. Properties of DFS

Assuming synchronized fixed-length quanta and a fixed set of runnable tasks, and the feasible share condition (equation 1), the following properties hold for DFS:

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

Lemma 2 *Given a set of feasible tasks, DFS is work-conserving in nature.*

DFS is similar to a P-fair scheduling algorithm proposed in [1]. We can show that under the ideal system assumption, DFS reduces to this algorithm. A proof of these properties can be found in [7].

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

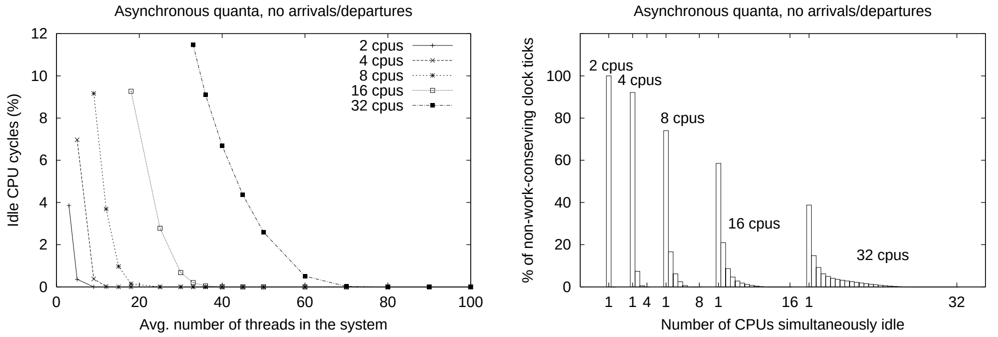
4. Ensuring Work-conserving Behavior in DFS

As indicated in Section 3.4, DFS is provably work-conserving under the assumption 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

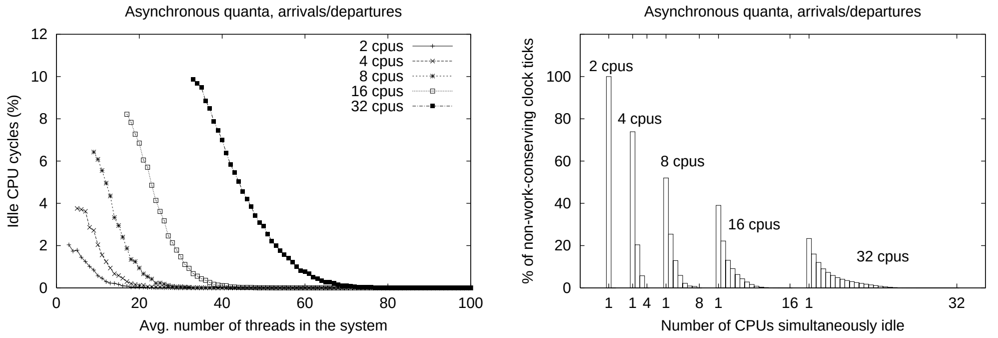
The methodology for our simulation study is as follows. We start with an idealized system that assumes a fixed task set and synchronized fixed length quanta. We then relax each assumption in turn and study the impact of doing so on the work-conserving nature (or lack thereof) of the scheduler. Specifically, the assumptions we relax incrementally are synchronous quanta in the system, fixed length quantum durations, and no arrivals and departures of tasks. 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. As a caveat, we have not used the tie-breaking rules outlined in the algorithm in our kernel implementation and simulation study, because, these rules are less meaningful when the ideal assumptions are relaxed.

To conduct our simulation study, we simulate multiprocessor systems with 2, 4, 8, 16 and 32 processors. We ini-



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

Figure 1. Effect of asynchronous quanta on the work-conserving behavior.



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

Figure 2. Effect of arrivals and departures on the work-conserving behavior.

tialize the system with a certain number of runnable tasks (Note that the actual number of tasks in a system can be much larger than that of runnable 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. As is true for an actual operating system, our simulations measure time in units of clock ticks. 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:

- *Ideal system:* As expected, our simulation results show 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 (Lemma 2) listed 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. As shown in Figure 1(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 the CPU cycles that are 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 1(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.

- *Variable length quanta:* Next, we let the quantum lengths vary but keep the number of tasks in the system fixed. Again, our simulations show that the system becomes non-work-conserving. The results obtained for this scenario (asynchronous variable-length quanta) are similar to that obtained in the previous scenario (asynchronous fixed-length quanta). This indicates that the asynchrony in scheduling is the primary cause for non-work-conserving behavior and variable length quanta does not substantially worsen this behavior. Since these results are similar to the previous scenario, we omit them for reasons of space.

- *Arrivals and departures:* Our final scenario adds arrivals and departures to the system. Again, we see that the system becomes non-work-conserving especially when the number of tasks is close to the number of processors (see Figure 2). 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 2(a) and 1(a)). We hypothesize that 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 have the opposite effect, seem to have a smaller impact on the non-work-conserving behavior. We hypothesize that this diminished effect is because work-conserving behavior is governed by the set of eligible tasks and departures typically do not affect this set (a departure affects the set of ineligible tasks, since that task would have been likely to be classified as ineligible if it hadn't departed).

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

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 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

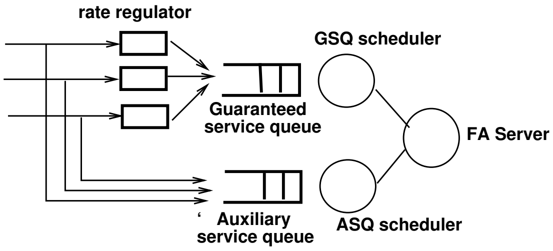


Figure 3. Fair Airport Scheduling Algorithm

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 3). 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 (Eq 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 ASQ tasks in the increasing order of 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 first reason is the simplicity of implementation, where we can simply reuse the data structures for DFS to implement the Fair Airport policy (see section 6 for details). Secondly, note that, scheduling tasks in order of start tags is equivalent to using *Start time fair queueing* [11], a scheduling algorithm that has known fairness and delay properties for uniprocessors. Thus, choosing this scheduling policy has the potential of providing 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 cache. 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 instance (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 cache effectiveness and the overall system performance [18].

Observe that the basic DFS algorithm uses internally generated deadlines (Eq 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.

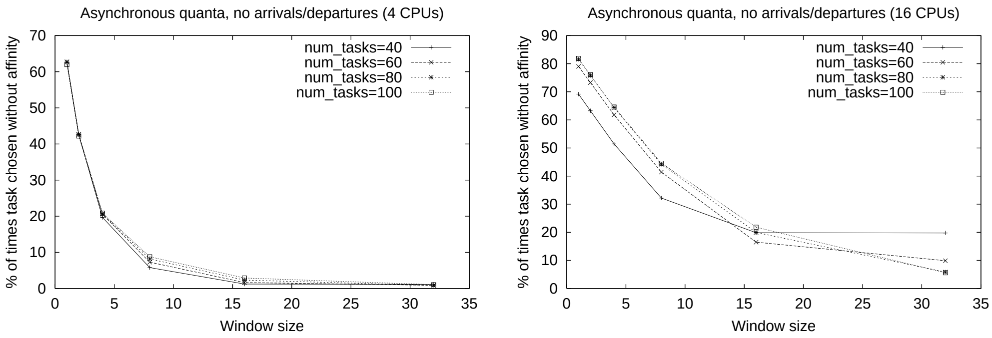
A second approach to account for processor affinities is to employ a single global run queue and use a more sophisticated metric for making scheduling decisions. Recall that the basic DFS algorithm stamps each eligible task with a deadline (Eq 7). Rather than using deadlines to schedule tasks, we define a new metric referred to as *goodness*. The goodness of a task is a function of its deadline and its affinity for a processor. In the simplest case, the goodness G can be defined as

$$G = d + \alpha \cdot \mathcal{A} \tag{8}$$

where d denotes the deadline of the task, α is a positive constant and $\mathcal{A}$ represents its affinity for a processor ($\mathcal{A}$ is 0 for the processor that it ran on last and 1 for all other processors). Thus α represents the penalty for scheduling a task on a different processor. The scheduler then picks the task with the minimum goodness value.

Assuming that the basic DFS algorithm maintains a list of eligible tasks sorted on their deadlines, the scheduling algorithm would then need to compute the goodness of each task in this list before picking the task with the minimum goodness (since the goodness is a processor dependent metric, it is not possible to compute the goodness 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 goodness 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 that was previously scheduled on a processor. 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 performed simulation experiments to determine the effectiveness of using goodness 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 time the scheduler is successfully able to pick a task with an affinity for the processor and also the resulting unfairness in the allocation. Figure 4 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 number of processors ($\mathcal{W} = p$) to balance the tradeoffs of scheduling efficiency and processor affinity. We conducted some experiments which showed that this rule of thumb does not greatly increase unfairness — e.g., using this rule on a 4-processor system, tasks remain within one quantum of their due share for 83% of the time. These results indicate that using goodness can be an effective technique to handle processor affinities in small to medium multiprocessor systems (< 32 processors).



(a) 4-CPU system (b) 16-CPU system

Figure 4. Effect of Window size on Processor Affinity

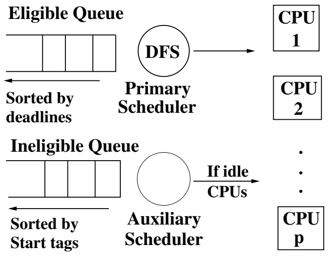


Figure 5. DFS-FA Scheduler

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

6. Implementation Considerations

We have implemented the basic 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`. 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 5). 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 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 the goodness metric 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 can them compute the goodness of the first $\mathcal{W}$ tasks in the eligible queue and schedule the task with the minimum goodness (see Eq 8). By choosing an appropriate value of α in Eq 8, 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 preliminary experimental evaluation. We conducted experiments to (i) demonstrate proportionate 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.

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.

7.1. Proportionate 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 6(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 6(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 the application from load in the system . Not surprisingly, the time-share scheduler is unable to provide such isolation.

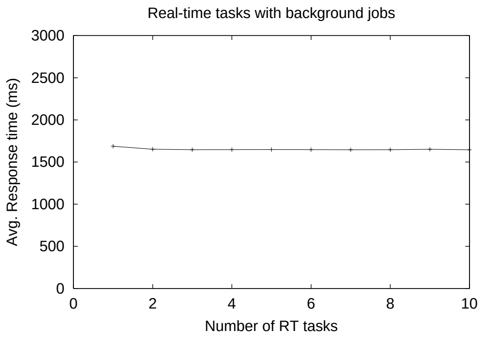


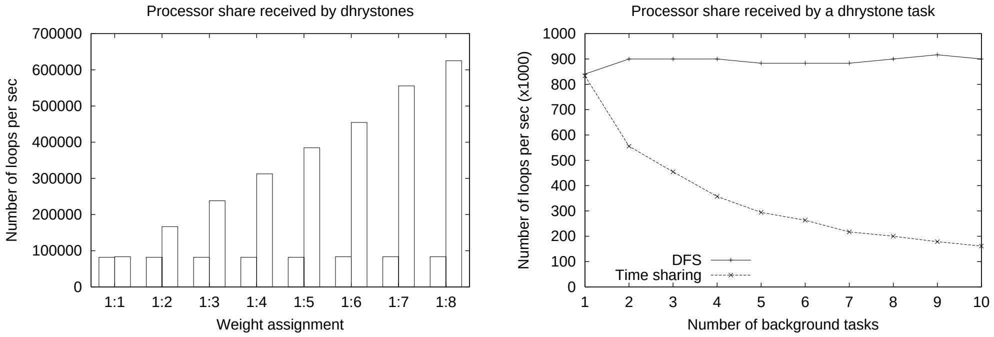
Figure 7. Performance of DFS when scheduling a mix of real-time applications.

These experiments demonstrate that while DFS-FA is no longer strictly P-fair, it nevertheless achieves proportionate allocation. In addition, it also manages to isolate applications from each other.

7.2. Impact on Real-Time and Multimedia Applications

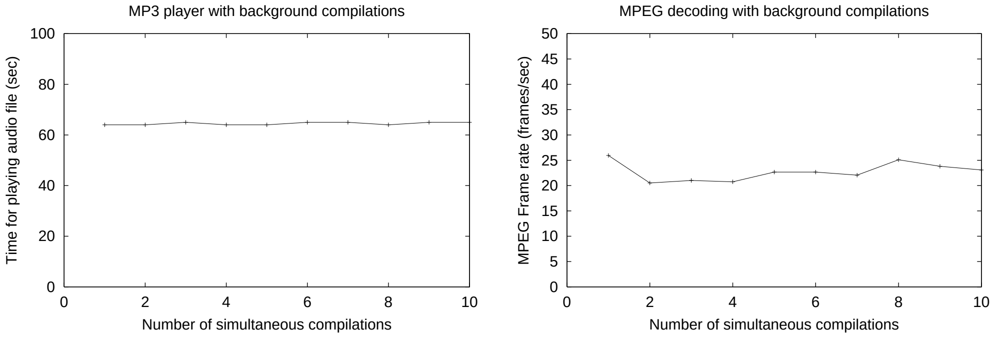
In the previous subsection, we demonstrated the desirable properties of DFS-FA using a synthetic, computation-intensive benchmark. Here, we demonstrate how DFS-FA can benefit real-time and multimedia applications. To do so, we first ran an experiment with a mix of *RT_tasks*, each of which emulates a 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 7, 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 8(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 8(b) shows that the frame rate of the mpeg decoder remains



(a) Proportionate Allocation (b) Application Isolation

Figure 6. Proportionate Allocation and Application Isolation with DFS-FA



(a) Streaming Audio (b) Streaming Video

Figure 8. Performance of multimedia applications.

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 at the disk. 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.

7.3. 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 1 summarizes the results we obtained. We report only those lmbench statistics that are relevant to the CPU scheduler. As can be seen from Table 1, 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).

Table 1. 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

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 P-fair schedulers 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 proportionate 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.

9. Acknowledgements

We would like to thank Krithi Ramamritham and Sanjoy Baruah for their valuable comments and discussions on earlier drafts of this paper. We would also like to thank the anonymous reviewers whose insightful comments and suggestions helped us improve the paper.

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] 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.

[3] M. Barabanov and V. Yodaiken. Introducing Real-Time Linux. *Linux Journal*, 34, February 1997.

[4] 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.

[5] 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.

[6] 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.

[7] A. Chandra, M. Adler, and P. Shenoy. Deadline Fair Scheduling: Bridging the Theory and Practice of Proportionate Fair Scheduling in Multiprocessor Systems. Technical Report TR00-38, Department of Computer Science, University of Massachusetts at Amherst, December 2000.

[8] R. 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. 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] M. B. Jones and J. Regehr. CPU Reservations and Time Constraints: Implementation Experience on Windows NT. In *Proceedings of the Third Windows NT Symposium, Seattle, WA*, July 1999.

[14] 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.

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

[16] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A Firm Real-Time System Implementation using Commercial Off-the-Shelf Hardware and Free Software. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1998.

[17] J. Stankovic, K. Ramamritham, D. Niehaus, M. Humphrey, and G. Wallace. The Spring System: Integrated Support for Complex Real-Time Systems. *Real-Time Systems Journal*, 16(2), May 1999.

[18] 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.

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

[20] R. West and K. Schwan. Dynamic Window-Constrained Scheduling for Multimedia Applications. In *IEEE International Conference on Multimedia Computing and Systems, Florence, Italy*, 1999.