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

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 schedulers in general-purpose operating systems. We find via a simulation study that characteristics of general-purpose operating systems such as the asynchrony in scheduling multiple processors, frequent arrivals and departures, and variable quantum durations can cause P-fair schedulers to be non-work-conserving. To overcome these limitations, we enhance DFS using the Fair Airport Scheduling framework to ensure work-conserving behavior at all times. We implement the resulting scheduler, referred to as DFS-FA, in the Linux kernel and demonstrate its performance on real workloads. Our experimental results show 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 the Fair Airport algorithm is a practical approach for scheduling tasks in multiprocessor operating systems.

1 Introduction

The growing popularity of the Internet has led to a proliferation of applications such as multi-player games, online virtual worlds, web hosting services, and continuous media streaming. These emerging applications tend to be compute-intensive and necessitate the use of large multiprocessor servers to accommodate their computing needs. Moreover, unlike conventional best-effort applications such as word processors and compilers, these applications require performance guarantees from the underlying operating system. To illustrate, an interactive multi-player game may require state updates to be propagated to all participants in 150 ms or less, while a streaming media server requires data to be accessed and transmitted to clients prior to its playback instant to ensure jitter-free playback. Since large servers run a mix of such demanding applications, a server operating system needs to employ resource management techniques that provide predictable performance to diverse application mixes.

In this paper, we present *Deadline Fair Scheduling (DFS)*, a proportional-share CPU scheduling algorithm for multiprocessor environments that achieves this objective. Whereas several proportional-share CPU scheduling algorithms have been proposed for single resource environments (e.g., uniprocessors), studies have shown that these schedulers do not perform well in multiprocessor environments [7, 17]. DFS is explicitly designed for multiprocessor environments and thereby addresses this limitation. The design and implementation of DFS has led to

several research contributions. First, DFS is based on the notion of *proportionate fairness* [4] and provides theoretically provable fairness guarantees. A particular focus of our work is to investigate practical issues in implementing proportionate-fair (P-fair) schedulers in general-purpose operating systems—prior work [1, 3, 4, 19] has focused mostly on analyzing the properties of P-fair schedulers. We find via a simulation study that characteristics of general-purpose operating systems such as the asynchrony in scheduling multiple processors, frequent arrivals and departures, and variable quantum durations can cause P-fair schedulers to be non-work-conserving. To ensure work-conserving behavior at all times based on these insights, we combine DFS with a work-conserving algorithm to derive DFS-FA—a new scheduler that belongs to the Fair Airport class of scheduling algorithms.[1] To the best of our knowledge, this is the first CPU scheduling algorithm that is based on the concept of fair airport scheduling. The synthesis of proportionate fairness and fair airport scheduling to achieve efficient, proportional-share allocation in general-purpose computing environments is a key contribution of our work. This synthesis results in several properties especially suited for server operating systems: (i) it allows a minimum share of the processor bandwidth to be assigned to each application, (ii) it performs fair, fine-grained allocation of bandwidth to applications based on their specified shares, (iii) it isolates applications from one another, thereby preventing overloaded or misbehaving applications from affecting other applications, (iv) it is work conserving in nature, and (v) it is computationally efficient to enable a practical implementation.

We have implemented DFS-FA in the Linux operating system and have made the source code available to the research community.[2]. We experimentally demonstrate the efficacy of our scheduler using numerous applications and benchmarks. Our results show that DFS-FA can achieve proportionate allocation, application isolation and work conserving behavior, albeit at a slight increase in scheduling overhead. We conclude from these results that combining a proportionate-fair scheduler such as DFS with the Fair Airport algorithm is a practical approach for scheduling tasks in 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 system model. We present the deadline fair scheduling algorithm in Section 4. In Section 5, we show how to combine DFS with the fair airport scheduling framework. Section 6 present the details of our implementation in Linux. Section 7 presents our experimental results. Finally, Section 8 presents some concluding remarks.

2 Fair Proportional-Share Scheduling: Basic Concepts

Emerging applications such as multi-player games and streaming media have timing constraints and require performance guarantees from the operating system. In the simplest case, such *soft real-time applications*[3] can be supported either by treating them as hard real-time or best-effort applications. Neither approach is satisfactory, since the former approach results in poor resource utilization due to the worst-case assumptions made by hard real-time schedulers,

[1] A fair airport scheduling algorithm typically combines a non-work-conserving scheduler with a work-conserving scheduler to achieve the best properties of both classes of algorithms [14].

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

[3] Like hard real-time applications, a soft real-time application has timing constraints and requires performance guarantees from the operating system, but unlike hard real-time applications, an occasional violation of these constraints does not result in incorrect execution or catastrophic consequences.

whereas the latter approach results in severely degraded performance in the presence of transient overloads or dynamically fluctuating workloads. Several resource management mechanisms have been developed to explicitly deal with soft real-time applications [2, 10, 12, 15, 16, 18, 20, 21, 24]. 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. Proportional-share schedulers can differ based on (i) the exact mechanism employed to specify resource shares (i.e., rates) and (ii) the mechanism employed to enforce these allocations at a fine time-scale.

The resource share (rate) associated with an application can be specified either in relative terms or absolute terms. In the former approach, each application is assigned a weight and resources are allocated in proportion to these weights. Thus, an application with weight w_i is allocated $(w_i / \sum_j w_j)$ fraction of the resource [6, 9, 13, 21, 25]. Weight-based allocation is relative since the actual fraction allocated to an application depends on the weights assigned to other applications. In the latter approach, each application is assigned an absolute fraction f_i, or a tuple (x_i, y_i) that indicates x_i units of time be allocated to the application every y_i units [4, 16, 23]. Thus, resources desired by an application are specified in absolute terms and are independent of other applications. To be feasible, such an absolute allocation approach requires that the total allocation should not exceed resource capacity (i.e., $\sum_j f_j \leq 1$ or $\sum_j \frac{x_j}{y_j} \leq 1$). It has been observed that relative allocation using weights and reservation-based absolute allocation are duals of each other [22].

Having specified the resource share of each application, a resource management mechanism must then enforce these allocations on a fine time-scale. The degree to which a scheduler can do so is referred to as its *fairness* property. Whereas proportional-share schedulers with several different fairness properties have been proposed, most of these schedulers can be categorized based on two different notions of fairness.

- *GPS fairness:* Generalized processor sharing (GPS) is an idealized algorithm that allocates bandwidth to applications in proportion to their weights. GPS ensures that, for any time interval $[t_1, t_2)$, the CPU service received by a thread i that is assigned weight w_i satisfies

$$\forall j, \quad \frac{A_i(t_1, t_2)}{A_j(t_1, t_2)} \geq \frac{w_i}{w_j} \tag{1}$$

  provided that thread i is continuously runnable in the entire interval [21]. A scheduling algorithm that satisfies this condition is said to be GPS-fair. It is easy to show that Equation 1 implies proportionate allocation of processor bandwidth.[4] GPS assumes that processor bandwidth can be allocated to applications using infinitesimally small quanta. In practice, a CPU scheduling algorithm employs finite duration quanta to amortize context switch overheads. Consequently, practical GPS-based fair schedulers such as weighted fair queuing (WFQ) [9, 21], self-clocked fair queuing (SCFQ) [11] and start-time fair queuing (SFQ) [13] do not strictly satisfy Equation 1 but strive to minimize the unfairness in their allocations by closely approximating GPS. Note that, generalized processor sharing assumes a single resource (uniprocessor) environment. Recently we have extended the notion of GPS fairness to multiprocessor environments by introducing a new idealized algorithm referred to as *generalized multiprocessor sharing (GMS)* (see [7] for details).

[4] This can be observed by summing Equation 1 over all runnable threads j, which yields $A_i(t_1, t_2) \cdot \sum_j w_j \geq w_i \cdot \sum_j A_j(t_1, t_2)$. Since $\sum_j A_j(t_1, t_2)$ is the total processor bandwidth allocated to all threads in the interval, we can substitute it by the quantity $p \cdot \mathcal{C} \cdot (t_2 - t_1)$, where $\mathcal{C}$ is the capacity of a processor. Hence, we get $A_i(t_1, t_2) \geq \frac{w_i}{\sum_j w_j} \cdot p \cdot \mathcal{C} \cdot (t_2 - t_1)$. Thus each thread receives processor bandwidth in proportion to its instantaneous weight w_i.

- *Proportionate fairness:* Proportionate fairness (P-fairness) is based on the idea of proportionate progress—each application is allocated processor bandwidth such that its progress is proportional to its weight. Specifically, proportionate fairness requires that, after T quantums, a continuously running application with weight w_i should have received between $\lfloor \alpha_i \cdot T \rfloor$ and $\lceil \alpha_i \cdot T \rceil$ quanta, where $\alpha_i = w_i / \sum_j w_j$. P-fairness is a strong fairness property since it ensures that, at any instant, no application is more than one quantum away from its due share. Unlike GPS-fairness, which assumes infinitely small quanta, P-fairness assumes finite fixed-duration quanta. In practice, however, blocking or I/O events may cause an application to relinquish the processor before it has used up the entire allocated quantum, and hence, quantum durations can vary from one quantum to another. Consequently, P-fairness is also an idealized notion of fairness applicable to a system where the quantum duration is always fixed.

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 idles a processor so long as there are runnable threads in the system. Non-work-conserving schedulers, on the other hand, can let idle a processor even in the presence of runnable threads. To see why this can happen, consider a a non-work-conserving proportional-share scheduler that allocates x_i units to a task every y_i units. The scheduler can achieve the requested allocation by assigning x_i units to this task every y_i units; if there are no other runnable tasks, then the processor will idle for the remaining $(y_i - x_i)$ time units. In contrast, a work-conserving scheduler will allocate these $(y_i - x_i)$ time units to the task to improve resource utilization. Thus, a work-conserving proportional-share scheduler treats the shares allocated to a process 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, most schedulers employed in general-purpose operating systems tend to be work-conserving in nature.

In this paper we focus on the design of a proportional-share P-fair scheduler for multiprocessor environments. A particular focus of our work is to design such a scheduler for general-purpose operating systems that support a mix of soft real-time and best-effort environments. Specifically, we require that the scheduler be work-conserving to achieve good resource utilization and that it perform proportional-share allocation even in the presence of variable quantum lengths and frequent arrivals and departures. Other desirable properties include the ability to achieve performance isolation for applications and computational efficiency to enable an efficient implementation. In what follows, we first present our system model and then present a proportional-share P-fair scheduler that achieves these objectives.

3 System Model

Consider a p-processor server that services n runnable tasks. Note that, the actual number of tasks in the system is typically larger; the remaining tasks are assumed to be blocked on I/O or synchronization events. In such a scenario, the CPU scheduler must decide which of these n tasks to schedule on the p processors. Each task scheduled by the CPU scheduler is allocated a quantum of duration q_{max}; a thread may voluntarily relinquish the processor before its allocated quantum ends due to a blocking event. Thus, the actual duration of a quantum depends on whether a thread utilizes its entire allocated quantum or blocks before the quantum expires. Consequently, in a typical multiprocessor

readjust(array $w[1..n]$**, int** i**, int** p**)**
begin
if($\frac{w[i]}{\sum_{j=i}^{n} w[j]} > \frac{1}{p}$)
begin
readjust($w[1..n]$,$i+1$,$p-1$)
$sum = \sum_{j=i+1}^{n} w[j]$
$w[i] = \frac{sum}{p-1}$
end
end.

Figure 1: The weight readjustment algorithm: The algorithm is invoked with an array of weights sorted in decreasing order. Initially, $i = 1$; p denotes the number of processors, and t denotes the number of runnable threads. If a thread violates the feasibility constraint, then the algorithm is recursively invoked for the remaining threads and the remaining processors. Each infeasible weight is then adjusted by setting its requested processor share to $1/p$.

system, quantums on different processors are *neither synchronized with each other, nor do they have a fixed duration*. Due to the asynchronous nature of quantums on different processors, each processor is assumed to independently invoke the scheduler whenever its current quantum ends.

Each task in the system is assigned a weight w_i; the CPU scheduler must then ensure that the task is allocated $\frac{w_i}{\sum_j w_j}$ fraction of the fraction of the bandwidth on p processors. Observe that an individual task can execute on only one processor at any time, and hence, can consume no more than the bandwidth of a single processor. Consequently, only those weight assignments in which each task requests a fraction no greater than $1/p$ are feasible. That is,

$$\forall i, \quad \frac{w_i}{\sum_{j=1}^{n} w_j} \leq \frac{1}{p} \tag{2}$$

We assume that a user can assign any arbitrary weight to a task. Hence, it is possible that a task may request more bandwidth than it can consume (i.e., $\frac{w_i}{\sum_j w_j} > \frac{1}{p}$) and cause the resulting weight assignment to be infeasible. Even if the initial weight assignment is carefully chosen to be feasible, blocking or wakeup events may cause the resulting set of runnable threads to have an infeasible weight assignment.[5] Consequently, the CPU scheduler must somehow reconcile the presence of infeasible weights. To do so, we have recently proposed a weight readjustment algorithm that, given a set of infeasible weights, translates it to the "closest" set of feasible weights [7]. The weight readjustment algorithm proceeds by examining the fraction requested by each task. The share of each task that requests more than $1/p$ is reset to exactly $1/p$ (the maximum it can consume); weights of tasks that request feasible shares are left unchanged. By invoking the weight readjustment algorithm every time the weight assignment becomes infeasible (which can potentially happen every time the set of runnable threads changes due to an arrival, departure, blocking, or wakeup event), the CPU scheduler can ensure that all scheduling decisions are always based on a set of feasible weights. As a convenience to the reader, we present the weight adjustment algorithm in Figure 1; see [7] for details.

Assuming the above environment, in what follows, we present a proportional-share P-fair CPU scheduling algorithm for multiprocessor servers.

[5]To illustrate, consider a dual-processor server that runs three tasks that are assigned weights $w_1 = w_2 = 1$ and $w_3 = 2$. The initial weight assignment is feasible since $w_i \leq 1/2$. However, the weight assignment becomes infeasible if task 1 blocks (since $w_3 = 2/3 > 1/2$).

4 Deadline Fair Scheduling

Consider a p-processor system that runs n tasks (n can vary over time due to arrivals and departures). Each task in the system is assigned a weight w_i. Let us assume that the weight readjustment algorithm translates the original weight assignment $w_1, w_2, \ldots, w_n$ into a new weight assignment $\phi_1, \phi_2, \ldots, \phi_n$, such that the new weights are feasible (the new weights may be identical to the original weights if the initial assignment was feasible). Given these weights, the deadline fair scheduling algorithm allocates $(\phi_i / \sum_j \phi_j)$ fraction of the total processing bandwidth to each task. DFS achieves these allocations based on the notion of proportionate fairness. To see how this is done, we first present the intuition behind the algorithm and then provide the precise details.

Conceptually, DFS schedules each task periodically, where the period of each task is proportional to its weight ϕ_i. DFS uses an eligibility criteria to ensure that each task runs *at most* once in each period and uses internally generated deadlines to ensure that each task run *at least* once in each period. The eligibility criteria makes each task eligible at the start of its period; once an eligible task runs on a processor, it 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 to ensure that the task runs by the end of its period. DFS schedules eligible tasks in *earliest deadline first* order to ensure each tasks receives its share before the end of its period. Together the eligibility criteria and the deadlines allow each task to receive processor bandwidth in proportion to its weight, 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 server that services three tasks with weights $w_1 = 2$, $w_2 = w_3 = 1$. The requested allocation can be achieved by running the first task continuously on one processor and alternating between other two tasks on the other processor. To see how this can be done using periods and deadlines, assume that the period of the first task is 1 and that of the other two tasks is 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. 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 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 has a period of 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. Thus, DFS achieves proportionate allocation by running task 1 continuously on one processor and alternating between the other two tasks on the other processor. Figure 2 illustrates this scenario.*

To intuitively understand how the eligibility criteria and deadlines are determined, for simplicity, let us assume that the quantum length=1, that each task is always run for an entire quantum, and that there are no arrivals or departures of tasks into the system. 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 quantums, and so on. With these assumptions, to maintain P-fairness, we require that for all times t and tasks i,

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

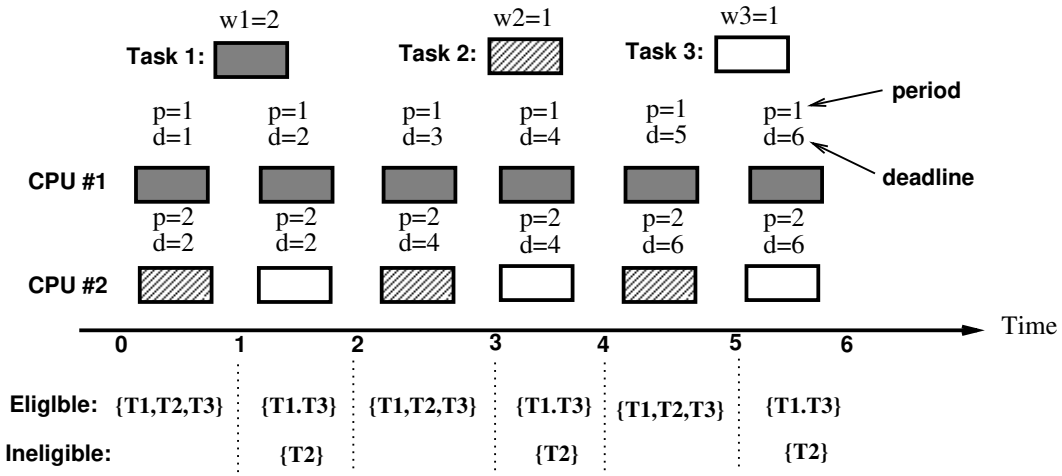


Figure 2: Use of deadlines and periods to achieve proportionate allocation.

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

$$m_i(t) + 1 \leq \left\lceil \frac{(t+1)p\phi_i}{\sum_{j=i}^{n} \phi_j} \right\rceil . \tag{3}$$

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

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

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

$$t' = \left\lceil (m_i(t) + 1) \frac{\sum_{j=i}^{n} \phi_j}{p\phi_i} \right\rceil . \tag{4}$$

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 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. If task i receives CPU service for a time interval of length q, then at the end of that period of service, S_i is incremented as $S_i = S_i + \frac{q}{\phi_i}$. In GPS-based algorithms, 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 3, 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 maximum possible weighted CPU service received by task i at the end of the next quantum where task i is run. If i may be run for the entire quantum, then $F_i = S_i + \frac{1}{\phi_i}$. However, if task i blocked during the last quantum it was run, and will only be run for some fraction of a quantum the next time it is run, and so F_i may also be smaller. Substituting $F_i = (m_i(t) + 1)/\phi_i$ into Equation 4, we see that the deadline for task i becomes $t' = \left\lceil \frac{\sum_{j=i}^{n} \phi_j}{p} F_i \right\rceil$.

Having defined the intuition for our algorithm, we next define the precise DFS algorithm as follows:

- Each task in the system is associated with a weight w_i, a start tag S_i and a finish tag F_i. Let ϕ_i denote the weight of a task as computed by the weight readjustment algorithm. 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

$$S_i = \begin{cases} \max(S_i, v) & \text{if the thread just woke up} \\ S_i + \frac{q}{\phi_i} & \text{if the thread is run on a processor} \end{cases} \tag{5}$$

  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.

- 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,

$$v = \max\left(v, \frac{\sum_{j=1}^{n} \phi_j \cdot S_j}{\sum_{j=1}^{n} \phi_j}\right) \tag{6}$$

  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 Criteria:* A task is eligible if it satisfies the following condition.

$$\frac{S_i \phi_i}{q_{max}} + 1 \leq \left\lceil \phi_i \left(\frac{v}{q_{max}} + \frac{p}{\sum_j \phi_j} \right) \right\rceil . \tag{7}$$

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

$$\left\lceil \frac{F_i}{q_{max}} \cdot \left(\frac{\sum_{j=1}^{n} \phi_j}{p} \right) \right\rceil \tag{8}$$

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, the deadline for such a task is increased by the ceiling operation, and thus it is given preference.

- Rule 2: If multiple tasks satisfy rule 1, then pick the task with the minimum value of

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

  Intuitively, this is the task that is expected to have the smallest deadline among all currently tied tasks the next time it becomes eligible.

Any further ties are broken arbitrarily. These tie-breaking rules are required by the proof from [1] that DFS is P-fair in the simplified case where there are no arrivals or departures, and every task always runs for a full quantum.

4.1 Properties of DFS

DFS is similar to a P-fair scheduling algorithm called Early Release Fair Scheduling that was recently proposed [1]. In fact, with no arrivals and departures and synchronized fixed-length quanta, it can be shown that DFS reduces to early release fair scheduling. Hence, all properties of early release scheduling also hold for DFS (under these assumptions). Since the properties of Early Release Fair Scheduling have been analyzed in detail, in this paper we provide only a brief listing of DFS properties and refer the reader to [1] for detailed proofs. Proofs for DFS use a similar methodology as [1] except that they are formulated using the notion of virtual time. Assuming synchronized fixed-length quanta and a fixed set of runnable tasks, the following properties hold for DFS:

- Given a set of tasks with feasible weights (as computed by the weight readjustment algorithm of Section 3), there always exists a schedule such that no eligible task misses its deadline. It follows from this property that DFS always generates a P-fair schedule.

- Given a set of tasks with feasible weights, DFS is work conserving in nature.

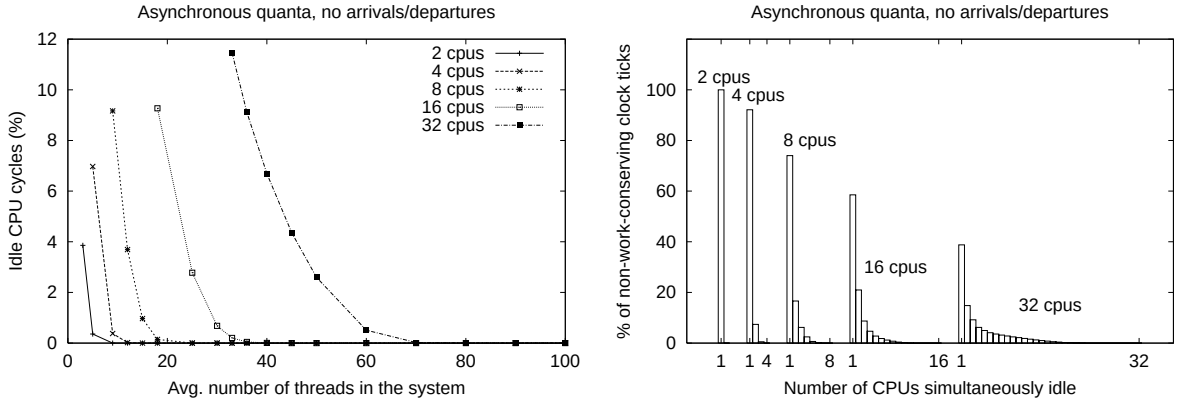
Next, we examine the behavior of DFS in a general-purpose operating system environment where the above restrictions do not hold.

4.2 Behavior of DFS in a General-purpose Operating System Environment

In the previous section, DFS was shown to have theoretically provable properties under the assumption of a fixed set of tasks and synchronized fixed-length quanta. However, as explained in Section 3, in a typical multiprocessor system, quantums on different processors are neither synchronized with each other, nor do they have a fixed duration. Moreover, the set of runnable tasks varies depending on arrivals/departures or blocking/wakeup events. In this section, we examine the impacts of these factors on the behavior of DFS. There are two issues to consider: (i) what is the impact of these factors on P-fairness and proportionate allocation? and (ii) what is their impact on the work conserving nature? We defer the first issue to Section 7 and consider only the second issue. In particular, we examine if the above factors can cause DFS to become non-work-conserving, since it may mark certain runnable tasks as ineligible and be left with fewer eligible tasks than processors (causing one or more processors to idle even when there are runnable tasks in the system). We conduct a simulation study to examine if such a scenario occurs and to what extent. We start with an ideal system where the set of tasks is fixed and quantums are synchronized and of fixed length. We add asynchrony to this system by allowing each processor to independently invoke the scheduler when its current quantum ends (the quantum duration and the set of tasks is kept fixed). Next, we let the quantum lengths vary, while keeping the number of tasks fixed. Finally, we allow arrivals and departures in the system. 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. The goal of our study is to determine if DFS is a suitable CPU scheduling algorithm for general-purpose multiprocessor systems. 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 general-purpose operating system without any modifications. In contrast, if the system becomes non-work-conserving for significant durations, then DFS will need to be enhanced to make it a practical choice for scheduling multiprocessor systems.

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 weight of each task is chosen randomly from a uniform distribution and the weight readjustment algorithm is employed to ensure that the weight assignment is feasible at all times. As in the case of 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 weight assignments). We obtain the following results from our study:

- *Ideal system:* As expected, our results showed that DFS is always work-conserving in an ideal system where the set of tasks is fixed and quantums are synchronized and of fixed length, which conforms to the theoretical properties listed in Section 4.1.



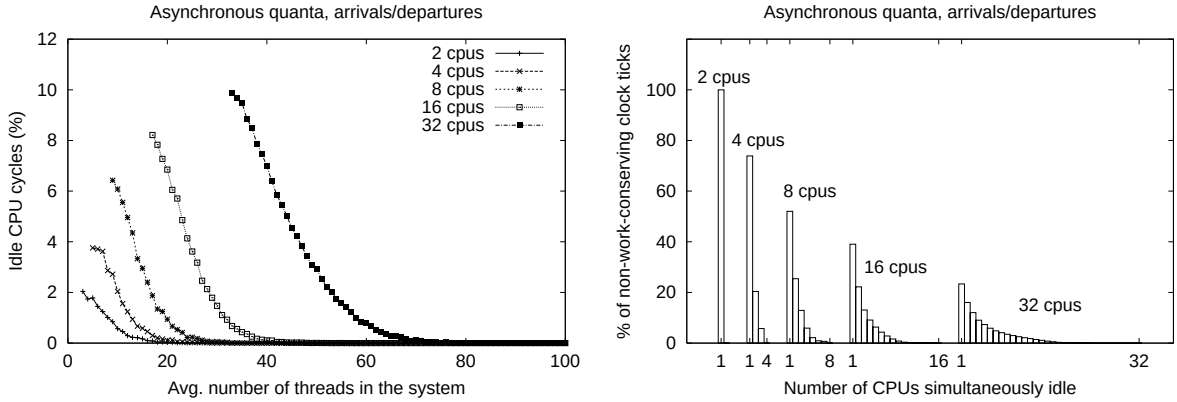
(a) Variation with number of tasks (b) Number of simultaneously idle processors

Figure 3: Effect of asynchronous quanta on the work-conserving behavior.

- *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 3(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 be 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 reduces the chances of the system becoming non-work-conserving. Figure 3(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 can 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 nearly identical 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 have a negligible impact on this behavior. We omit these results due to their similarity with the previous scenario and space constraints.

- *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 4). Interestingly, we find that the fraction of CPU cycles that are wasted *decreases*slightly as compared to the previous two scenarios (see Fig 3(a) and 4(a)). This is because each arrival introduces an additional eligible task into the system, causing an idle processor (if one exists) to schedule the newly arrived task (without new 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 system behavior. This is because our eligibility criteria causes a currently running task to become ineligible once it relinquishes the processor; hence, departure of such a task does not affect the set of eligible tasks.



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

Figure 4: Effect of arrivals and departures on the work-conserving behavior.

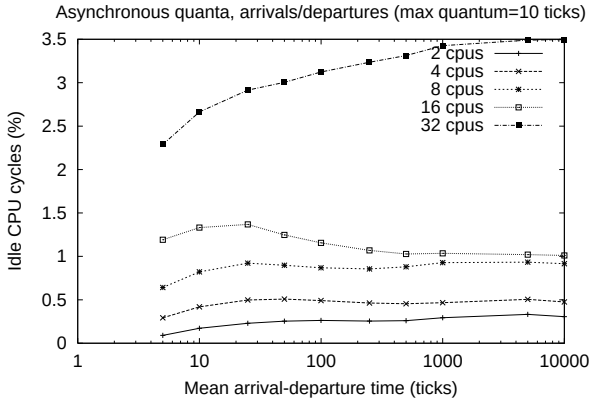


Figure 5: Effect of the arrival/departure rate on the work-conserving behavior.

Figure 5 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 times implies fewer arrivals, which then reduces the probability that an idle processor schedules a newly arriving task).

We conclude from our simulations that DFS can exhibit non-work-conserving behavior in a general-purpose multiprocessor system. Since the fraction of CPU cycles that can be wasted can be as large as 10-12%, this indicates the need to enhance DFS with a policy that reallocates idle processor bandwidth to tasks that are ineligible but runnable (and thereby improve overall system utilization). Since our results have shown that multiple processors can be simultaneously idle in the non-work-conserving state, this residual allocation of idle bandwidth is itself a multiprocessor scheduling problem (since the policy must decide how best to schedule a set of ineligible, runnable tasks on a dynamically varying number of idle processors). In what follows, we enhance our deadline fair scheduling algorithm to achieve this objective.

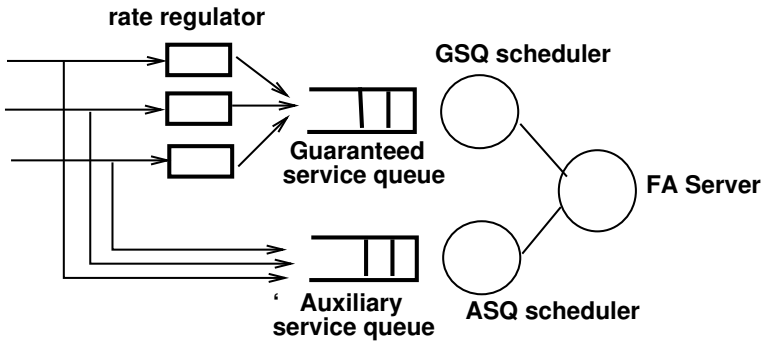


Figure 6: Fair Airport Scheduling Algorithm

5 Combining DFS with Fair Airport Scheduling Algorithms

The previous section demonstrated the need to enhance DFS with a policy that allocates bandwidth on idle processors to tasks that are runnable but deemed ineligible by DFS. Such a policy can ensure that a processor will not idle so long as there are runnable tasks in the system and improves resource utilization. To achieve this objective, we employ a concept referred to as *fair airport scheduling* that was previously proposed for scheduling network packets at a router [8, 14]. In a fair airport scheduler, each runnable task joins a rate regulator and an Auxiliary Service Queue (ASQ) (see Figure 6). Once a task passes through a rate regulator, it joins a Guaranteed Service Queue (GSQ) if it has not been serviced by then. Observe that, the combined scheduler always gives priority to the GSQ over the ASQ—the GSQ scheduler gets to schedule tasks 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 tasks in the guaranteed service and auxiliary service queues.

Our instantiation of fair airport, refered to as DFS-FA, employs DFS to schedule tasks in the guaranteed services queue. The rate regulator for each task is simply its eligibility criteria; the criteria ensures that a tasks joins the guaranteed services queue only once in each period. We use the auxiliary service queue to service tasks when the guaranteed service queue becomes empty (i.e., when DFS becomes non work-conserving). In such a scenario, the ASQ schedules tasks from the ASQ on each idle processor (observe that tasks in the ASQ are runnable but ineligible). Since our simulation results indicated that multiple processors could become idle simultaneously, the scheduling algorithm employed to service the ASQ should be able to achieve proportionate allocation of this residual bandwidth in a multiprocessor environment. To ensure that residual bandwidth is allocated fairly among competing tasks (in proportion to their weights), we employ *surplus fair scheduling (SFS)*, a multiprocessor GPS-based scheduling algorithm that was proposed recently. SFS is a work-conserving proportional-share scheduling algorithm, and hence, can achieve fair allocation of residual (i.e., idle) bandwidth to tasks [7]. Whereas we use fairness as the criteria for allocating residual bandwidth in the ASQ, it is possible to employ other policies (e.g., prioritized allocation) and use appropriate work conserving schedulers that achieve such allocations.

Thus, the combined algorithm uses DFS to service tasks in the guaranteed service queue, and uses SFS to allocate residual bandwidth to tasks in the auxiliary queue, and hence is work conserving at all times.

Table 1: System calls used for controlling weights of tasks

Syscall	Description
`int setweight(int which, int who, int weight)`	Set weight of a process, process group or user
`int getweight(int which, int who)`	Return weight of a process, process group or user

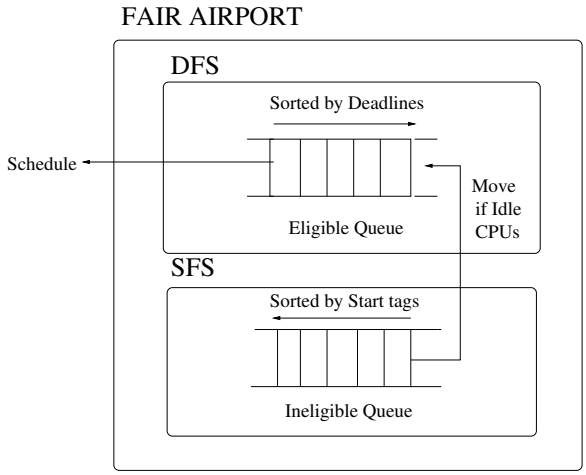


Figure 7: DFS-FA Scheduler

6 Implementation Considerations

We have implemented the DFS-FA algorithm in the Linux kernel version 2.2.14 (source code is for our implementation is available from our web site).

In this section, we present the details of our implementation and the various design decisions and considerations we have taken into account.

6.1 Data Structures and Algorithm

In our implementation, we have implemented DFS-FA as a replacement of the standard Linux time-sharing scheduler. The modified kernel uses the DFS-FA scheduler for making all scheduling decisions. Thus every time a task relinquishes its CPU, the DFS-FA scheduler is called to select a task to run on that CPU. Each task in the system is assigned a weight which is used by the scheduler to determine its CPU share. By default, each task is assigned a weight of 1. Tasks can dynamically change or query their weights using two new system calls, `setweight` and `getweight`. These system calls are described in Table 1. Their interface is very similar to the Linux system calls `setpriority` and `getpriority`.

In our implementation of DFS-FA, we maintain three queues of runnable tasks. The first queue, called the *weight queue*, consists of all runnable tasks in the decreasing order of their weights. This queue is required for efficient

readjustment of non-feasible weights as described in section 3. The second queue, called the *eligible queue*, consists of those runnable tasks which are determined to be eligible by the DFS algorithm. The scheduler can pick a task to run only from this queue. The tasks in this queue are sorted in the increasing order of their deadlines, as DFS schedules them in the *earliest deadline first (EDF)* order. The last queue, called the *ineligible queue*, consists of all the remaining (ineligible) runnable tasks. These tasks are maintained in the increasing order of their start tags, as this is the order in which tasks become eligible in the algorithm. Further, our implementation of the Fair Airport algorithm employs the *surplus fair scheduling (SFS)* to keep the scheduler work-conserving [7]. An efficient implementation of SFS requires an additional queue that maintains all ineligible tasks in order of surplus values; in the absence of this queue, the entire ineligible queue may need to be scanned by SFS to determine the task with the least surplus. Figure 7 shows the main components of the DFS-FA scheduler and how they fit in with each other.

The actual scheduler works as follows. Whenever a task's quantum expires or it blocks for I/O or departs, the Linux kernel calls the DFS-FA scheduler. The scheduler first updates the start tag and finish tag of the task relinquishing the CPU, and deletes it from the runnable task queues if it is no longer runnable. 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, it moves them from the ineligible queue to the eligible queue in order of their deadlines. If the task relinquishing the CPU is still runnable, it is put into the appropriate queue based on its new start and finish tags. Then the scheduler just picks the task at the head of the eligible queue for execution. Finally, the readjustment algorithm is also invoked whenever the set of tasks changes (due to arrivals/departures or blocking/wakeup events), or when a task weight changes due to a call to `setweight`.

6.2 Implementation Complexity

In this section, we describe the implementation complexity of the DFS-FA algorithm.

- *New arrival or wakeup event:* The scheduler first inserts the newly arrived/woken up task into the appropriate queues. These insertions are $O(\log n)$ operations. Further, insertions need to be done only in the weight queue and eligible queue as threads in the eligible and ineligible queues are mutually exclusive. Next, readjustment algorithm needs to be run which takes $O(p)$ time. Then the scheduler recomputes the virtual time, which is $O(n)$ if the readjustment algorithm changes the weights of existing tasks, else it is a no-op. Overall, the complexity of a new arrival or wakeup event is $O(n + p)$.

- *Departure or blocking event:* In this case, the departing/blocking task needs to be deleted from the various runnable queues. These deletions take constant time. Further, readjustment algorithm needs to be run, which is $O(p)$, and virtual time needs to be recomputed, which can be an $O(n)$ operation or a no-op. Overall complexity, therefore, is $O(n + p)$.

- *Scheduling:* If the set of tasks and their weights remain unchanged at a scheduling instant, the scheduler needs to do the following. First of all, it has to update the start tag and finish tag of the task relinquishing the CPU. This is an $O(1)$ operation. Then, it needs to do an incremental update of the virtual time, which is also a constant time operation. Next, it needs to move any newly eligible tasks from the ineligible queue to

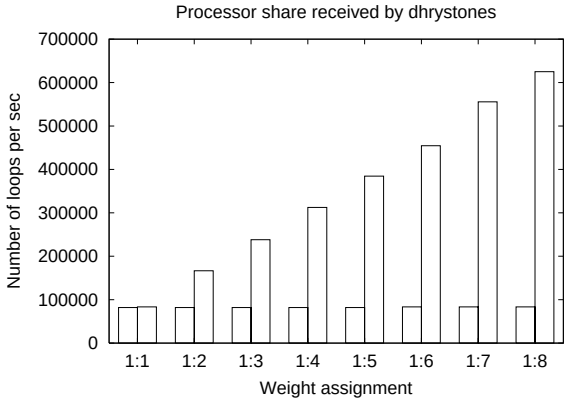


Figure 8: Proportionate CPU Allocation

the eligible queue. Scanning for the newly eligible tasks can be done in order, as the tasks in the ineligible queue are sorted by their start tags. Moving each of them is an $O(\log n)$ operation ($O(1)$ for deletion from the ineligible queue and $O(\log n)$ for insertion into the eligible queue). Finally, the scheduler just needs to pick the task at the head of the eligible queue for running, which is $O(1)$. Overall, the time complexity is $O(n \log n)$.

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. We used 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.

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 6.0. 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 : *dhrystone*, a compute-intensive benchmark for measuring integer performance, and *lmbench*, a benchmark that measures various aspects of operating system performance. Next, we describe the results of our experimental evaluation.

7.2 Proportionate Allocation

We first demonstrate that DFS-FA allocates processor bandwidth to applications in proportion to their weights. To show this property, we ran two *dhrystone* applications with relative weights of 1:1, 1:2, 1:3, 1:4, 1:5, 1:6, 1:7 and 1:8

Table 2: 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

in the presence of 20 background dhrystone applications (The background applications were required to ensure that weights were feasible at all times for the two main tasks). As can be seen from figure 8, the two applications receive processor bandwidth in proportion to their weights. Also, our measurements indicate that the system never becomes non-work-conserving during these experiments This demonstrates that DFS-FA is work-conserving at all times. We note that, while DFS-FA is no longer strictly P-fair (especially at times when it allocates residual (idle) bandwidth to ineligible tasks), it nevertheless achieves proportionate allocation. Thus, DFS-FS balances the conflicting goals of strict P-fair scheduling and work-conserving behavior, and hence, is a practical choice for multiprocessor systems.

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 2 summarizes the results we obtained. We report only those lmbench statistics that are relevant to the CPU scheduler. As can be seen from Table 2, 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 200 ms quantum duration used by the Linux kernel.

8 Conclusion

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 that characteristics of general-purpose operating systems such as the asynchrony in scheduling multiple processors, frequent arrivals and departures, and variable quantum durations can cause P-fair schedulers to be 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 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 concluded that combining a proportionate-fair scheduler such as DFS with the Fair Airport algorithm is a practical approach for scheduling tasks in multiprocessor operating systems.

Acknowledgements

We would like to thank Krithi Ramamritham for numerous discussions on the design of P-fair scheduling algorithms.

References

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

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

[4] 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.

[5] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. Fair On-Line Scheduling of a Dynamic Set of Tasks on a SingleResource. Technical Report TR-96-03, Department of Computer Sciences, The University of Texas at Austin, February 1996.

[6] J.C.R. Bennett and H. Zhang. Hierarchical Packet Fair Queuing Algorithms. In *Proceedings of SIGCOMM'96*, pages 143–156, August 1996.

[7] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors. Technical Report TR00-22, Department of Computer Science, University of Massachusetts at Amherst, April 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] S.J. Golestani. A Self-Clocked Fair Queueing Scheme for High Speed Applications. In *Proceedings of INFOCOM'94*, pages 636–646, April 1994.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprocessing in a Hard-Real Time Environment. *JACM*, 20:46–61, January 1973.

[18] 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.

[19] M. Moir and S. Ramamurthy. Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resources. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98), Madrid, Spain*, December 1998.

[20] 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.

[21] 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.

[22] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. In *Proceedings of the ACM/SPIE Conference on Multimedia Computing and Networking (MMCN'97), San Jose, CA*, pages 207–214, February 1997.

[23] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of Real Time Systems Symposium*, December 1996.

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

[25] H. Zhang and S. Keshav. Comparison of Rate-Based Service Disciplines. In *Proceedings of ACM SIGCOMM*, pages 113–121, August 1991.