A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems

Ion Stoica [*] Hussein Abdel-Wahab [†] Kevin Jeffay[‡] Sanjoy K. Baruah [§]
Johannes E. Gehrke [¶] C. Greg Plaxton [||]

Abstract

We propose and analyze a proportional share resource allocation algorithm for realizing real-time performance in time-shared operating systems. Processes are assigned a weight which determines a share (percentage) of the resource they are to receive. The resource is then allocated in discrete-sized time quanta in such a manner that each process makes progress at a precise, uniform rate. Proportional share allocation algorithms are of interest because (1) they provide a natural means of seamlessly integrating real- and non-real-time processing, (2) they are easy to implement, (3) they provide a simple and effective means of precisely controlling the real-time performance of a process, and (4) they provide a natural mean of policing so that processes that use more of a resource than they request have no ill-effect on well-behaved processes.

We analyze our algorithm in the context of an idealized system in which a resource is assumed to be granted in arbitrarily small intervals of time and show that our algorithm guarantees that the difference between the service time that a process should receive in the idealized system and the service time it actually receives in the real system is optimally bounded by the size of a time quantum. In addition, the algorithm provides support for dynamic operations, such as processes joining or leaving the competition, and for both fractional and non-uniform time quanta. As a proof of concept we have implemented a prototype of a CPU scheduler under FreeBSD. The experimental results shows that our implementation performs within the theoretical bounds and hence supports real-time execution in a general purpose operating system.

1 Introduction

Currently there is great interest in providing real-time execution and communication support in general purpose operating systems. Indeed, applications such as desktop videoconferencing, distributed shared virtual environments, and collaboration-support systems require real-time computation and communication services to be effective. At present the dominant approach to providing real-time support in a general purpose operating system is to embed a periodic thread or process model into an existing operating system kernel and to use a real-time scheduling algorithm such as rate-monotonic scheduling to schedule the processes. In such as system, aperiodic and non-real-time activities are typically scheduled either as background processes or through the use of a second-level scheduler that is executed quasi-periodically as a server process by the real-time scheduler.

In general, this framework can be quite effective at integrating real-time and non-real-time computing. However, we observe that this approach has yet to be embraced by the larger operating systems community. We believe that this is due in part to the rigid distinctions made between real-time and non-real-time. Real-time activities are programmed differently than non-real-time ones (e.g., as periodic tasks) and real-time activities receive hard and fast guarantees of response time (if admission control is performed). Non-real-time

[*] Supported by GAANN fellowship. Dept. of CS, Old Dominion Univ., Norfolk, VA 23529-0162 (stoica@cs.odu.edu).

[†] Supported by NSF grant CCR 95–9313857. Dept. of CS, Old Dominion Univ., Norfolk, VA 23529-0162 (wahab@cs.odu.edu).

[‡] Supported by grant from the IBM & Intel corps and NSF grant CCR 95–10156. Dpt. of CS, Univ. of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175, (jeffay@cs.unc.edu).

[§] Supported by NSF under Research Initiation Award CCR–9596282. Dept. of CS, Univ. of Vermont, Burligntom, VT 05405, (sanjoy@cs.uvm.edu).

[¶] Dpt. of CS, Univ. of Wisconsin-Madison, Madison, WI 53706-1685 (johannes@cs.wisc.edu).

[||] Supported by NSF grant CCR–9504145, and the Texas Advanced Research Program under grant No. ARP–93–00365–461. Dpt. of CS, Univ. of Texas at Austin, Austin, TX 78712-1188 (plaxton@cs.utexas.edu).

activities are subservient to real-time ones and receive no performance guarantees. While this state of affairs is entirely acceptable for many mixes of real-time and non-real-time activities, for many it is not. Consider the problem of supporting real-time videoconferencing on the desktop. This is clearly a real-time application, however, it is not one for which hard and fast guarantees of real-time performance are required. For example, it is easy to imagine situations in which one would like to explicitly degrade the performance of the videoconference (e.g., degrade the rate at which the video is displayed) so that other activities, such as searching a large mail database for a particular message, can complete quicker. Ideally, a general purpose operating system that supports real-time execution should not a priori restrict the basic tenor of performance guarantees that any process is capable of obtaining.

To address this issue we investigate an alternate approach to realizing real-time performance in time shared operating systems, namely the use of *proportional share resource allocation* algorithms for processor scheduling. In a proportional share allocation system, every process in the system is guaranteed to make progress at a well-defined, uniform rate. Specifically, each process is assigned a share of the processor — a percentage of the processor's total capacity. If a process's share of the processor is s then in any interval of length t, the process is guaranteed to receive $(s \times t) \pm \epsilon$ units of processor time where $0 \leq \epsilon \leq \delta$, for some constant δ. In a proportional share system, resource allocation is flexible and the share received by a process can be changed dynamically. In this manner a process's real-time rate of progress can be explicitly controlled.

Proportional share resource allocation algorithms lie between traditional general purpose and real-time scheduling algorithms. On the one hand, proportional share resource allocation is a variant of the *pure processor sharing* scheduling discipline, in which during each time unit each process receives $1/n$ of the processor's capacity, where n is the number of active processes. (Thus each process appears as it is making uniform progress on a virtual processor that has $1/n$ of the capacity of the physical processor.) On the other hand, traditional real-time scheduling disciplines for periodic tasks can be viewed as coarse approximations of proportional share allocation. For example, if a periodic task requires c units of processor time every p time units, then a rate-monotonic scheduler guarantees that for all $k \geq 0$, in each interval $[kp, (k+1)p]$, the periodic task will indeed receive a share of the processor equal to c/p. Specifically, in each of the above intervals, the process will receive $pc/p = c$ units of processor time.

Our proportional share resource allocation policy differs from traditional methods of integrating real- and non-real-time processes in that here all processes, real- and non-real-time, are treated identically. In a proportional share system, real-time (and non-real-time) processes can be implemented very much like traditional processes in a time-shared operating system. Thus in terms of the process model, no "special" support is required to support real-time computing — there is only one type of process. Moreover, like many scheduling algorithms used in time-shared systems, our algorithm allocates resources in discrete units or quanta which makes it easier to implement than traditional real-time policies which are typically event-driven and require the ability to preempt processes at potentially arbitrary points. In addition, because resources are allocated in discrete quanta in a proportional share system, one can better control (and account for) the overhead of the scheduling mechanism as well as tune the system to trade-off fine-grain, real-time control for low scheduling and system overhead. Finally, proportional share algorithms provide a natural means of uniformly degrading system performance in overload situations.

In this paper we present a proportional share scheduling algorithm and demonstrate that it can be used to ensure predictable real-time response to all processes. Section 2 presents our process model and formally introduces the concepts of a share and the requirement for predictable execution with respect to a share. Section 3 discusses related work in scheduling. Section 4 presents a deadline-based, virtual-time scheduling algorithm that is used to ensure processes receive their requested share of the processor. Section 5 introduces a key technical problem to be solved in the course of applying our algorithm, namely that of dealing with the dynamic creation and destruction of processes. Section 6 outlines the proof of correctness of our algorithm and Section 7 presents some experimental results using our proportional share system in the FreeBSD operating system, and demonstrates how "traditional" real-time processes such as periodic tasks can be realized in a proportional share system.

2 The Model

We consider an operating system to consist of a set of processes (real-time and non-real-time) that compete for a time shared resource such as a CPU or a communications channel. To avoid confusion with terminology used in the experimental section of the paper we use the term *client* to refer to computational entities (i.e., processes). A client is said to be *active* while it is competing for the resource, and *passive* otherwise. We assume that the resource is allocated in time quanta of size at most q. At the beginning of each time quantum a client is selected to use the resource. Once the client acquires the resource, it may use it either for the entire time quantum, or it may release it before the time quantum expires. Although simple, this model captures the basic mechanisms traditionally used for sharing common resources, such as processor and communication bandwidth. For example, in many preemptive operating systems (e.g., UNIX, Windows-NT), the CPU scheduler allocates the processing time among competing processes in the same fashion: a process uses the CPU until its time quantum expires or another process with a higher priority becomes active, or it may voluntarily release the CPU while it is waiting for an event to occur (e.g., an I/O operation to complete). As another example, consider a communication switch that multiplexes a set of incoming sessions on a packet-by-packet basis. Since usually the transmission of a packet cannot be preempted, we take a time quantum to be the time required to send a packet on the output link. Thus, in this case, the size q of a time quantum represents the time required to send a packet of maximum length.

Further, we associate a *weight* with each client that determines the relative *share* of the resource that it should receive. Let w_i denote the weight associated to client i, and let $\mathcal{A}(t)$ be the set of all clients active at time t. We define the (instantaneous) share $f_i(t)$ of an active client i at time t as

$$f_i(t) = \frac{w_i}{\sum_{j \in \mathcal{A}(t)} w_j}. \qquad (1)$$

If the client's share remains constant during a time interval $[t, t + \Delta t]$, then it is entitled to use the resource for $f_i(t)\Delta t$ time units. In general, when the client share varies over time, the service time that client i should receive in a *perfect fair* system, while being active during a time interval $[t_0, t_1]$, is

$$S_i(t_0, t_1) = \int_{t_0}^{t_1} f_i(\tau) d\tau \qquad (2)$$

time units. The above equation corresponds to an ideal fluid-flow system in which the resource can be granted in arbitrarily small intervals of time[1]. Unfortunately, in many practical situations this is not possible. One of the reasons is the overhead introduced by the scheduling algorithm itself and the overhead in switching from one client to another: taking time quanta of the same order of magnitude as these overheads could drastically reduce the resource utilization. Another reason is that some operations cannot be interrupted, i.e., once started they must complete in the same time quantum. For example, once a communication switch begins to send a packet of one session, it cannot serve any other session until the entire packet is sent. As another example, a process cannot be preempted while it is in a critical section. Thus, in the first example we can choose the size of a quantum q as being the time required to send a packet of maximum length, while in the second example we can choose q as being the maximum duration of a critical section.

Due to quantization, in a system in which the resource is allocated in discrete time quanta it is not possible for a client to always receive *exactly* the service time it is entitled to. The difference between the service time that a client should receive at a time t, and the service time it *actually* receives is called service time *lag* (or simply lag). Let t_0^i be the time at which client i becomes active, and let $s_i(t_0^i, t)$ be the service time the client receives in the interval $[t_0^i, t]$ (here, we assume that client i is active in the entire interval $[t_0^i, t]$). Then the lag of client i at time t is

$$lag_i(t) = S_i(t_0^i, t) - s_i(t_0^i, t). \qquad (3)$$

Since the lag quantifies the allocation accuracy, we use it as the main parameter in characterizing our proportional share algorithm. In particular, in Section 6 we show that our proportional share algorithm (1) provides bounded lag for all clients, and that (2) this bound is optimal in the sense that it is not possible to develop an algorithm that affords better bounds. Together, these properties indicate that our algorithm will provide real-time response guarantees to clients and that with respect to the class of proportional share algorithms, these guarantees are the best possible.

3 Related Work

Tijdeman was one of the first to formulate and analyze the proportional share allocation problem [15]. The original problem, an abstraction of diplomatic protocols, was stated in terms of selecting a union chairman every year, such that the accumulated number of chairmen from each state (of the union) to be proportional to its weight. As shown in [2], Tijdeman's results can be easily applied to solve the proportional share allocation problem. In the general setting, the resource is allocated in fixed time quanta, while the clients' shares may change at the beginning of every time quantum. In this way dynamic operation can be easily accommodated. Tijdeman proved that if the clients' shares are known in advance there exists a schedule with the lag bound less or equal to $1 - 1/(2n-2)$, where n represents

[1] A similar model was used by Demers *et al* [4] in studying fair-queuing algorithms in communication networks.

the total number of clients. (Note that when $n \rightarrow \infty$ the lag bound approaches unity.) Although he gives an optimal algorithm for the *static* case (i.e., when the number of clients does not change over time), he does not give any explicit algorithm for the dynamic case. Furthermore, we note that, even in the general setting, the problem formulation does not accommodate fractional or non-uniform quanta.[2]

Recently, the proportional share allocation problem has received a great deal of attention in the context of operating systems and communication networks. Our algorithm is closely related to weighted fair queueing algorithms previously developed for bandwidth allocation in communication networks [4, 5, 10], and general purpose proportional share algorithms, such as stride scheduling [17, 18]. Demers, Keshav, and Shenker were the first to apply the notion of fairness to a fluid-flow system that models an idealized communication switch in which sessions are serviced in arbitrarily small increments [4]. Since in practice a packet transmission cannot be preempted, the authors proposed an algorithm, called Packet Fair Queueing (PFQ), in which the packets are serviced in the order in which they would finish in the corresponding fluid-flow system (i.e., in the increasing order of their virtual deadlines). By using the concept of *virtual time*, previously introduced by Zhang [19], Parekh and Gallager have analyzed PFQ when the input traffic stream conforms to the *leaky-bucket* constraints [10, 11]. In particular, they have shown that no packet is serviced T_{max} latter than it would have been serviced in the fluid-flow system, where T_{max} represents the time to transmit a packet of maximum size. However, as shown in [3, 13, 18], the lag bound can be as large as $O(n)$, where n represents the number of active sessions (clients) in the system. Moreover, in PFQ the virtual time is updated when a client joins or leaves the competition in the ideal system, and not in the real one. This requires one to maintain an additional event queue, which makes the implementation complex and inefficient. As a solution, Golestani has proposed a new algorithm, called Self-Clocked Fair Queueing (SCFQ), in which the virtual time is updated when the client joins/leaves the competition in the real system, and not in the idealized one [5]. Although this scheme can be more efficiently implemented, this does not come for free: the lag bounds increase to within a factor of two of the ones guaranteed by PFQ.

Recently, Waldspurger and Weihl have developed a new proportional share allocation algorithm, called stride scheduling [17, 18], which can be viewed as a cross-application of fair queueing to the domain of processor scheduling. Stride scheduling relies on the concept of *global pass* (which is similar to virtual time) to measure the work progress in the system. Each client has an associated *stride* that is inversely proportional to its weight, and a *pass* that measures the progress of that client. The algorithm allocates a time quantum to the client with the lowest pass, which is similar to the PFQ policy. However, by grouping the clients in a binary tree, and recursively applying the basic stride scheduling algorithm at each level, the lag is reduced to $O(\log n)$. Moreover, stride scheduling provides support for both uniform and non-uniform quanta.

Goyal, Guo and Vin have proposed a new algorithm, called Start-time Fair Queueing (SFQ), for hierarchically partitioning of a CPU among various application classes [6]. While this algorithm supports both uniform and non-uniform quanta, the delay bound (and implicitly the lag) increases linearly with the number of clients. However, we note that when the number of clients is small, in terms of delay, this algorithm can be superior to classical fair queueing algorithms.

In contrast to the above algorithms, by making use of both virtual eligible times and virtual deadlines, the algorithm we develop herein achieves constant lag bounds, while providing full support for dynamic operations. We note that two similar algorithms were independently developed (in parallel to our original work [13]) by Bennett and Zhang in the context of allocating bandwidth in communication networks [3], and by Baruah, Gehrke and Plaxton in the context of processor scheduling for fixed time quanta [2]. In addition to introducing the concept of virtual eligible time (which was also independently introduced in [2] and [3]) our work makes several unique key contributions.

First, by "decoupling" the request size from the size of a time quantum we generalize the previous known theoretical results [10]. Moreover, our analysis can be easily extended to preemptive systems, as well. For example, we can derive lag bounds for a fully preemptive system, by simply taking time quanta to be arbitrarily small. Similarly, by taking the size of a time quantum to be the maximum duration of a critical region, we can derive lag bounds for a preemptive system with critical regions. Finally, this decoupling gives a client possibility of trading between allocation accuracy and scheduling overhead (see Section 6).

Second, we address the problem of a client leaving the competition *before* using the entire service time it has requested. This is an important extension since in an operating system it is typically not possible to predict *exactly* how much service time a client will use for

[2] The difference between fractional and non-uniform quanta is that while in the first case the fraction from the time quantum (that the client will actually use) is assumed to be known in advance, in the non-uniform quanta case this fraction is not known.

the next request. We note that this problem does not occur and consequently has not been addressed in the context of network bandwidth allocation; in this case, the length of a message and therefore its transmission time is assumed to be known upon its arrival. The only previous known algorithms that address this problem are lottery and stride scheduling [17, 18]. However, the lag bounds guaranteed by stride scheduling are as large as $O(n)$, where n represents the number of active clients (being a randomized algorithm, lottery does not guarantee tight bounds). In comparison, our algorithm (described next) guarantees optimal lag bounds of one time quantum.

Third, we propose a new approximation scheme for maintaining virtual time, in which update operations are performed when the events (e.g., client leaving, joining) occur in the real system, and not in the ideal one. This simplifies the implementation and eliminates the need to keep an event queue. It is worth mentioning that unlike other previous approximations [5], ours guarantees optimal lag bounds.

Besides the class of fair queuing algorithms, a significant number of other proportional share algorithms have recently been developed [1, 9, 12, 16]. Although none of them guarantees constant lag bounds in a *dynamic* system, we note that the PD algorithm of Baruah, Gehrke, and Plaxton [1] achieves constant lag bounds in a *static* system.

The idea of applying fair queueing algorithms to processor scheduling was first suggested by Parekh in [11]. Waldspurger and Weihl were the first to actually develop and implement such an algorithm (stride scheduling) for processor scheduling [17, 18].[3] Finally, to our best knowledge we are the first to implement and to test a proportional share scheduler which guarantees constant lag bounds.

4 The EEVDF Algorithm

In order to obtain access to the resource, a client must issue a *request* in which it specifies the duration of the service time it needs. Once a client's request is fulfilled, it may either issue a new request or become passive. For uniformity, throughout this paper we assume that the client is the sole initiator of the requests For flexibility we allow the requests to have any duration. Note that a client may request the same amount of service time by generating either fewer longer requests, or many shorter ones. For example, a client may ask for one minute of computation time either by issuing 60 requests with a duration of one second each, or by issuing 600 requests with a duration of 100 ms each. As we will show in Section 6, shorter requests guarantee better allocation accuracy, while longer requests decrease system overhead. This affords a client the possibility of trading between allocation accuracy and scheduling overhead.

We formulate our scheduling algorithm in terms of the behavior of an ideal, fluid-flow system that executes clients in a *virtual-time* time domain [19, 10]. Abstractly, the virtual fluid-flow system executes each client for w_i real-time time units during each virtual-time time unit. More concretely, virtual-time is defined to be the following function of real-time

$$V(t) = \int_0^t \frac{1}{\sum_{j \in \mathcal{A}(\tau)} w_j} d\tau. \tag{4}$$

Note that virtual-time increases at a rate inversely proportional to the sum of the weights of all active clients. That is, when the competition increases virtual-time slows down, while when the competition decreases it accelerates. Intuitively, the flow of virtual-time changes to "accommodate" all active clients in one virtual-time time unit. That is, the size of a virtual-time unit is modified such that in the corresponding fluid-flow system each active client i receives w_i real-time units during one virtual-time time unit. For example, consider two clients with weights $w_1 = 2$ and $w_2 = 3$. Then the rate at which virtual-time increases relative to real-time is $\frac{1}{w_1+w_2} = 0.2$, and therefore a virtual-time time unit equals five real-time units. Thus, in each virtual-time time unit the two clients should receive $w_1 = 2$, and $w_2 = 3$ time units.

Ideally we would like for our proportional share algorithm to approach the behavior of the virtual fluid-flow system. Thus, since in the fluid-flow system, at all points in time a client is best characterized by the service time it has received up to the current time, to compare our approach with the ideal, we must be able to compute the service time that a client should receive in the fluid-flow system. By combining Eq. (1) and (2) we can express the service time that an active client i should receive in the interval $[t_1, t_2)$ as

$$S_i(t_1, t_2) = w_i \int_{t_1}^{t_2} \frac{1}{\sum_{j \in \mathcal{A}(\tau)} w_j} d\tau. \tag{5}$$

Once the integral in the above equation is computed, we can easily determine the service time that *any* client i should receive during the interval $[t_1, t_2)$, by simply multiplying the client's weight by the integral's value. Next, from Eq. (5) and (4) it follows that

$$S_i(t_1, t_2) = (V(t_2) - V(t_1)) w_i. \tag{6}$$

[3] We note that they have also applied stride scheduling to other shared resources, such as critical section lock accesses.

To better interpret the above equation consider a much simpler model in which the number of active clients is constant and the sum of their weights is one ($\sum_{i \in \mathcal{A}} w_i = 1$), i.e., the share of a client i is $f_i = w_i$. Then, in this model, the service time that client i should receive during an interval $[t_1, t_2)$ is simply $S_i(t_1, t_2) = (t_2 - t_1)w_i$. Next, notice that by replacing the real times t_1 and t_2 with the corresponding virtual-times $V(t_1)$ and $V(t_2)$ we arrive at Eq. (6). Thus, Eq. (6) can be viewed as a generalization for computing the service time $S_i(t_1, t_2)$ in a *dynamic* system — one in which clients are dynamically joining and leaving the competition.

Our scheduling algorithms uses measurements made in the virtual-time domain to make scheduling decisions. For each client's request we define an *eligible time* e and a *deadline* d which represent the starting and finishing time respectively for the request's service in the corresponding fluid-flow system. Let t_0^i be the time at which client i becomes active, and let t be the time at which it initiates a new request. Then, a request becomes eligible at a time e when the service time that the client should receive in the corresponding fluid-flow system, $S_i(t^0, e)$, *equals* the service time that the client has already received in the real system, $s_i(t^0, t)$, (i.e., $S_i(t_0^i, e) = s_i(t_0^i, t)$). Note that if at time t client i has received more service time than it was supposed to receive (i.e., $lag_i(t) < 0$), then it will be the case that $e > t$ and hence the client should wait until time e before the new request becomes eligible. In this way a client that has received more service time than its share is "slowed down", while giving the other active clients the opportunity to "catch up". On the other hand, if at time t client i has received less service time than it was supposed to receive (i.e., its lag is positive), then it will be the case that $e < t$, and therefore the new request is immediately eligible at time t. By using Eq. (6) the virtual eligible time $V(e)$ is

$$V(e) = V(t_0^i) + \frac{s_i(t_0^i, t)}{w_i}. \tag{7}$$

Similarly, the *deadline* of the request is chosen such that the service time that the client should receive between the eligible time e and the deadline d *equals* the service time of the new request, i.e., $S_i(e, d) = r$, where r represents the length of the new request. By using again Eq. (6), we derive the virtual deadline $V(d)$ as

$$V(d) = V(e) + \frac{r}{w_i}. \tag{8}$$

Notice that although Eq. (7) and (8) give us the virtual eligible time $V(e)$ and the virtual deadline $V(d)$, they do not *necessarily* give us the values of the real times e and d! To see why, consider the case in which e is larger than the current time t. Then e cannot be computed exactly from Eq. (4) and (7), since we do *not* know how the slope of the virtual-time mapping will vary in the future (it changes dynamically while clients join and leave the competition). Therefore we will formulate our algorithm in terms of *virtual* eligible times and deadlines and not of the real times. With this, the Earliest Eligible Virtual Deadline First (EEVDF) algorithm can be simply stated as follows:

EEVDF Algorithm. *A new quantum is allocated to the client which has the eligible request with the earliest virtual deadline.*

Since EEVDF is formulated in terms of virtual-times, in the remaining of this paper we use ve and vd to denote the virtual eligible time and virtual deadline respectively, whenever the corresponding real eligible time and the deadline are not given. Let $r^{(k)}$ denote the length of the k^{th} request made by client i, and let $ve^{(k)}$ and $vd^{(k)}$ denote the virtual eligible time and the virtual deadline associated to this request. If each client's request uses the *entire* service time it has requested, then by using Eq. (7) and (8) we obtain the following recurrence which computes both the virtual eligible time and the virtual deadline of each request:

$$ve^{(1)} = V(t_0^i), \tag{9}$$

$$vd^{(k)} = ve^{(k)} + \frac{r^{(k)}}{w_i}, \tag{10}$$

$$ve^{(k+1)} = vd^{(k)}. \tag{11}$$

Next, we consider the more general case in which the client does not use the entire service time it has requested. Since a client never receives more service time than requested, we need to consider only the case when the client uses the resource for less time than requested. Let $u^{(k)}$ denote the service time that client i actually receives during its k-th request. Then the only change in Eq. (9)–(11), will be in computing the eligible time of a new request. Specifically, Eq. (11) is replaced by

$$ve^{(k+1)} = ve^{(k)} + \frac{u^{(k)}}{w_i}. \tag{12}$$

Example. To fix the ideas, let us take a simple example (see Figure 1). Consider two clients with weights $w_1 = w_2 = 2$ that issue requests with lengths $r_1 = 2$, and $r_2 = 1$, respectively. We assume that the time quantum is of unit size ($q = 1$) and that client 1 is the first one which enters competition at real time $t_0 = 0$. Thus, according to Eq. (9) and (10) the virtual eligible time for the first request of client 1 is $ve = 0$, while its virtual deadline is $vd = 1$. Being the single client that has an outstanding eligible request, client 1 receives the first quantum. At real time $t = 1$, client

2 enters the competition. Since during the interval $[0, 1)$ the only active client in the system is client 1, from Eq. (4), the value of virtual-time at real-time 1 is $V(1) = \int_0^1 \frac{1}{w_1} d\tau = 0.5$. Thus, virtual-time increases at half the rate of real-time. In this way, in an ideal system, during every virtual-time time unit, client 1 receives $w_1 = 2$ real time units. Next, after the second client enters the competition, the rate of virtual-time slows down further to $\frac{1}{w_1 + w_2} = 0.25$. Hence, in the ideal system, during one virtual-time time unit, each client will receive 2 real time units (since $w_1 = w_2 = 2$). Next, assume that client 2 issues its first request just before the second quantum is allocated. Then at real time $t = 1$ there are two pending requests: one of client 1 with the virtual deadline 1 (which waits for another time quantum to fulfill its request), and one of client 2 which has the same virtual deadline, i.e., 1. In this situation we arbitrarily break the tie in favor of client 2, which therefore receives the second quantum. Since this quantum fulfills the current request of client 2, client 2 issues immediately, at real time 3 (virtual-time 1), a new request. From Eq. (11) and (10) the virtual eligible time and the virtual deadline of the new request are 1 and 1.5, respectively. Thus, at real time $t = 2$ (virtual-time 0.75) the single eligible request is the one of client 1, which therefore receives the next quantum. Further, at real time $t = 3$ (virtual-time 1) there are again two eligible requests: the one of client 2 that has just become eligible, and the new request issued by client 1. Since the virtual deadline of the second client's request (1.5) is earlier than the one of the first client (2), the fourth quantum is allocated to client 2. Further, Figure 1 shows how the next four quanta are allocated.

Note the uniform progress of the two clients in Figure 1. Although the uniformity is perfect in this contrived example, we show in Section 6 that in fact the deviation of a client's progress from the perfectly uniform rate (i.e., its rate of progress in the ideal fluid-flow system) is bounded and that these bounds are the best possible. This shows that for a given quanta q, the EEVDF algorithm provides the best possible guarantees of real-time progress.

5 Fairness in Dynamic Systems

In this section we address the issue of *fairness* in dynamic systems. Throughout this paper, we assume that a dynamic system provides support for client joining and leaving the competition[4]. To understand the main issues of implementing dynamic operations, first recall that the client's lag represents (see Eq. (3)) the difference between the service time that the client *should* receive and the service time it *has* actually received. An important property of the EEVDF algorithm is that at *any* time the sum of the lags of all active clients is zero (see Lemma 2 in [14]). Thus, if a client leaves the competition with a negative lag (i.e., after receiving more service time than it was supposed to), the remaining clients should have received *less* service time than they were entitled to. In short, a gain for one client translates into a loss for the other active clients. Similarly, when a client with positive lag leaves, this translates into a gain for the remaining clients. The main question here is how to distribute this loss/gain among the remaining clients. In [13] we answered this question by distributing it in *proportion* to the clients' weights. In the remaining of this section we show that the same answer is obtained by approaching the problem from a different angle.

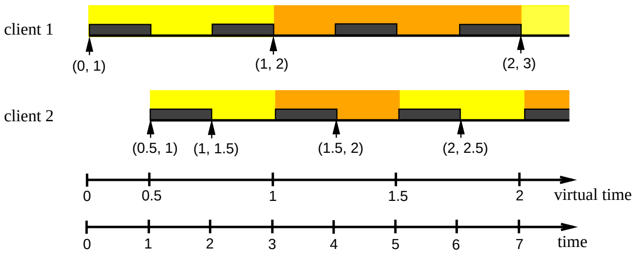

Figure 1. *An example of EEVDF scheduling involving two clients with equal weights $w_1 = w_2 = 2$. All the requests generated by client 1 have length 2, and all of the requests generated by client 2 have length 1. Client 1 becomes active at time 0 (virtual-time 0), while client 2 becomes active at time 1 (virtual-time 0.5). The vertical arrows represent the times when the requests are initiated (the pair associated to each arrow represents the virtual eligible time and the virtual deadline of the corresponding request). The shaded regions in the background show the the durations of servicing successive requests (of the same client) in the fluid-flow system.*

The basic observation is that this problem does not occur as long as a client with zero lag leaves the competition, because there is nothing to distribute. Since in the corresponding fluid-flow system the lag of any client is *always* zero, a simple solution would be to consider the time when the client leaves to be the time when it leaves in the corresponding fluid-flow system, and

[4] Note that with these two operations, changing a client's weight can be easily implemented as a leave followed by a re-join operation [13].

not in the real system. Unfortunately, this solution has two major drawbacks. First, in many situations, such as scheduling incoming packets in a high speed networking switch, maintaining the events in the fluid-flow system is too expensive in practice [5]. Second and more important, this solution assumes implicitly that the service time that a client will use is known in advance. While this is generally true in the case of the communication switch, where the length of a message (and consequently its service time) is assumed to be known when the packet arrives, in the processor case this is not always possible. To see why this is a potential problem, consider the previous example (see Figure 1) in which the first client leaves the competition after using only 1.1 time-units of the second request, i.e., at time 6.1 in the real system and the corresponding virtual time 1.775. However, according to Eq. (12), in the ideal system the client should complete its service and therefore leave the competition at virtual time 1.55 ($= ve^{(2)} + u^{(2)}/w_1$), which in our example corresponds to the real time 5.2. Unfortunately, since at this point we do not know for how long client 1 will continue to use the resource (we know only that it has made a request for *two* time-units of execution and has actually executed for only one time unit) we cannot update the virtual time correctly!

Next, we present our solution to this problem for a dynamic system in which the following two (reasonable) restrictions hold: (1) all the clients that join the competition are assumed to have zero lag, and (2) a client has to leave the competition as soon as it is finished using the resource (i.e., when a client terminates it is not allowed to remain in the system). We consider two cases depending on whether the client's lag is negative or positive. From Eq. (3), (4), (6) it follows that the client's lag increases as long as the client receives service time, and decreases otherwise. Thus, when a client with negative lag wants to leave, we can simply delay that client (without allocating any service time to it) until its lag becomes zero. This can be simply accomplished by generating a dummy request of zero length. However, note that since a request cannot be processed before it becomes eligible, and since the virtual eligible time of the dummy request is equal to its deadline (see Eq. (8)), this request cannot be processed earlier than its deadline. In this way, we have reduced the first case to the second one, in which the client leaving the competition has a positive lag. Our solution is based on the same idea as before: the client is delayed until its lag becomes zero.

For clarity, consider the example in Figure 2(a), where three clients become simultaneously active. Next, suppose that at time t_1, client 1 decides to leave the competition while having a positive lag. Then the client will be simply delayed, while continuing to receive service time, until its lag becomes zero, i.e., until time t_1'. If we assume that the slope of virtual-time with respect to real-time does not change between t_1 and t_1', then from Eq. (5) and (6) we obtain $S_1(t_1, t_1') = (V(t_1') - V(t_1))w_1 = w_1(t_1' - t_1)/(w_1 + w_2 + w_3)$. Further, by using Eq. (3) and (5), and the fact that $s_1(t_1, t_1') = t_1' - t_1$ we can compute the virtual-time at t_1' as

$$V(t_1') = V(t_1) + \frac{lag_1(t_1)}{w_2 + w_3} \tag{13}$$

The main drawback of this approach is that client 1 continues to receive service time between t_1 and t_1', although it does not need it (since it has already finished using the resource)! Thus, this service time will be wasted, which is unacceptable. Our solution is to simply let any client with positive lag leave immediately, while correctly updating the value of virtual-time to account for the change (see Figure 2(b)). In this way the virtual-times corresponding to the times when a client decides to leave and when it actually leaves are the *same* in both systems. More precisely consider a client k leaving the competition at time t_k with a positive lag (i.e., $lag_k(t_k) > 0$). Then, by generalizing Eq. (13), the value of virtual-time is updated as follows

$$V(t_k) = V(t_k) + \frac{lag_k(t_k)}{\sum_{j \in \mathcal{A}(t_k) \setminus \{k\}} w_j}, \tag{14}$$

where $\mathcal{A}(t_k)$ represents the set of all active clients just *before* client k leaves. For example, in Figure 2(b) $\mathcal{A}(t_1) = \{1, 2, 3\}$. Thus, $\mathcal{A}(t_k) \setminus \{k\}$ represents the set of all active clients just *after* client k leaves the competition. Further note that according to Eq. (3) the lag of any remaining client $i \in \mathcal{A}(t_k) \setminus \{k\}$ changes to

$$lag_i(t_k) = lag_i(t_k) + w_i \frac{lag_k(t_k)}{\sum_{j \in \mathcal{A}(t_k) \setminus \{k\}} w_j}. \tag{15}$$

Thus the lag of client i is *proportionally* distributed among the *remaining* clients, which is consistent with our interpretation of fairness in dynamic systems, i.e., any gain or loss is proportionally distributed among the remaining clients.

Since virtual-time is updated only when the events actually occur in the real system (as opposed to when they occur in the ideal one), the EEVDF algorithm can be easily and efficiently implemented. Even in a system in which the service times are known in advance, it is theoretically possible to update virtual-time as in the ideal system, however, in practice this is hard to achieve. Mainly, this is because we need to implement an event queue which has to balance the trade-off between timer granularity and scheduling overhead. As

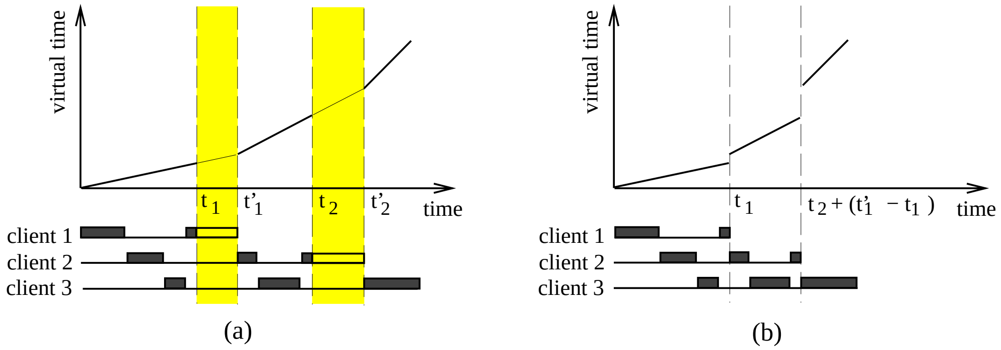

Figure 2. *Three clients become active at the same time, after which client 1 and client 2, both with positive lags, leave the competition. In (a) clients are allowed to leave only after their lags become zero; in (b) clients are allowed to leave immediately. The shaded regions in (a) represents the time intervals during which the system allocates service time to the clients until their lags become zero. In both cases the virtual-time just before a client wants to leave and just after it has actually left are equal.*

we have shown in [13] all the basic operation required to implement the EEVDF algorithm, i.e., inserting and deleting a request, and finding the eligible request with the earliest deadline can be implemented in $O(\log n)$, where n represents the number of active clients.

We note that in the worst case it may be possible that all the dummy requests occur at the same time. In this situation, the scheduler should perform $O(n)$ deletions before the next "real" request is serviced. Although, this is a potential problem in the case of a communication switch, where the selection of the next packet to be serviced is assumed to be done *during* the transmission of the current packet, it does not significantly increase the complexity of CPU scheduling. This is mainly because a processor, besides servicing the active clients (processes), also executes the scheduling algorithm, as well as other related operating system functions (e.g., starting a new process, or terminating an existing one). Consequently, in a complete model we need to account for these overheads anyway. A simple solution would be to charge each process for the related overheads. For example, the time to select the next process to receive a time quantum should be charged to that client. In this way, from the processor's perspective, a dummy request is *no* longer a 0-duration request since it should account at least for the scheduling overhead (and eventually for the process termination). In the current model we ignore these overheads, which, as the experimental results suggest (see Section 7), is an acceptable approximation for many practical situations. However, we plan to address this aspect in the future.

6 Fairness Results

The proportional share scheduling algorithm we have proposed executes clients at a precise rate. One can determine if a client has a desired real-time response time property by simply computing the amount of service time it is to receive during the interval(s) of time of interest using either Eq. (5) or (6). However, because service time is allocated in discrete quanta, this computation is off by the client's lag. Thus, in order to use our proportional share algorithm for real-time computing, we must demonstrate that the lag incurred by any client is bounded at all times. This is done next.

The problem is stated as that of demonstrating that the EEVDF algorithm is *fair* in the sense that all clients make progress according to their weights. By demonstrating that the lag of each client is bounded at all times we conclude that our algorithm is fair. Here we sketch the argument that lags are bounded. The complete proof of each result are given in the extended version of this paper [14].

Theorem 1 shows that any request is fulfilled no latter than q time units after its deadline in the corresponding fluid-flow system, where q represents the maximum size of a time quantum. Theorem 2 gives tight bounds for the lag of any client in a system in which all the clients that join and leave the competition have zero lags. Similarly, Theorem 3 gives tight bounds for the client's lag in a system in which a client with positive lag may leave at any time. Finally, as a corollary we show that in a dynamic system in which no client request is larger than the maximum size q of a time quantum the lag of any client is bounded by q. Moreover, this result is optimal with respect to *any* proportional share algorithm. We begin by defining formally the systems we are analyzing.

Definition 1 *A* **steady** *system (***S-system** *for short) is a system in which the lag of any client that joins, or leaves the competition is zero.*

The next definition is a formal characterization of the system described in Section 5 (see Figure 2(b)).

Definition 2 *A* **pseudo-steady** *system (***PS-system** *for short) is a system in which the lag of any client that joins is zero, and the lag of any client that leaves is positive. Moreover, when a client with positive lag leaves, the value of virtual-time is updated according to Eq. (14).*

The following theorem gives the upper bound for the maximum delay of fulfilling a request in an S-system. We note that this result generalizes a previous result of Parekh and Gallager [10] which holds for the particular case in which a request is no larger than a time quantum.

Theorem 1 *Let d be the deadline of the current request issued by client k in an S-system with quantum q, and let f be the actual time when this request is fulfilled. Then*

1) the request is fulfilled no later than $d + q$, i.e., $f < d + q$, and

2) if $f > d$, for any time $t \in [d, f)$, $lag_k(t) < q$.

The next theorem gives tight bounds for a client lag in an S-system.

Theorem 2 *Let r be the size of the current request issued by client k in an S-system with quantum q. Then the lag of client k at any time t while the request is pending is bounded as follows*

$$-r < lag_k(t) < max(r, q),$$

Moreover, these bounds are asymptotically tight.

Notice that the bounds given by Theorem 2 apply independently to each client and depend only on the lengths of their requests. While shorter requests offer a better allocation accuracy, the longer ones reduce the system overhead since for the same total service time fewer requests need to be generated. It is therefore possible to trade between accuracy and system overhead, depending on client requirements. For example, for a computationally intensive task it would be acceptable to take the length of the request to be on the order of seconds. On the other hand, in the case of a multimedia application we need to take the length of a request to be no greater than several tens of milliseconds, due to the delay constraints. Theorem 2 shows that EEVDF can accommodate clients with different requirements, while guaranteeing tight bounds for the lag of each client, which are independent of the other clients. As the next theorem shows this is not true for PS-systems. In this case the lag of a client can be as large as the maximum request issued by *any* client in the system.

Theorem 3 *Let r be the size of the current request issued by client k in a PS-system with quantum q. Then the lag of client k at any time t while the request is pending is bounded as follows*

$$-r < lag_k(t) < max(R_{max}, q),$$

where R_{max} represents the maximum duration of any request issued by any client in the system. Moreover, these bounds are asymptotically tight.

The following corollary follows directly from Theorems 2 and 3.

Corollary *If no request of client k is larger than a time quantum, then at any time t its lag is bounded as follows:*

$$-q < lag_k(t) < q.$$

Finally, we note that according to the following simple lemma (the proof can be found in [13]) the bounds given in the above corollary are optimal, i.e., they hold for *any* proportional share algorithm.

Lemma *Given any system with time quanta of size q and any proportional share algorithm, the lag of any client is asymptotically bounded by $-q$ and q.*

7 Experimental Results

As a proof of concept we have implemented a CPU scheduler prototype based on our EEVDF algorithm under FreeBSD v 2.0.5. All the experiments were run on a PC compatible with a 75 MhZ Pentium and 16 MB of RAM. The scheduler time slice (quantum), and the duration of any client's request were set to 10 ms.

Excepting the CPU scheduler, we did not alter the FreeBSD kernel.[5] Our scheduler coexists with the original FreeBSD scheduler [7]. All the processes that request proportional share or reservation services are assigned a reserved (user-level) priority, and are handled

[5] Indeed, the fact that we could perform these experiments on top of a largely unmodified general purpose operating system indicates the good fit between proportional share resource allocation scheme we advocate and general purpose operating system design.

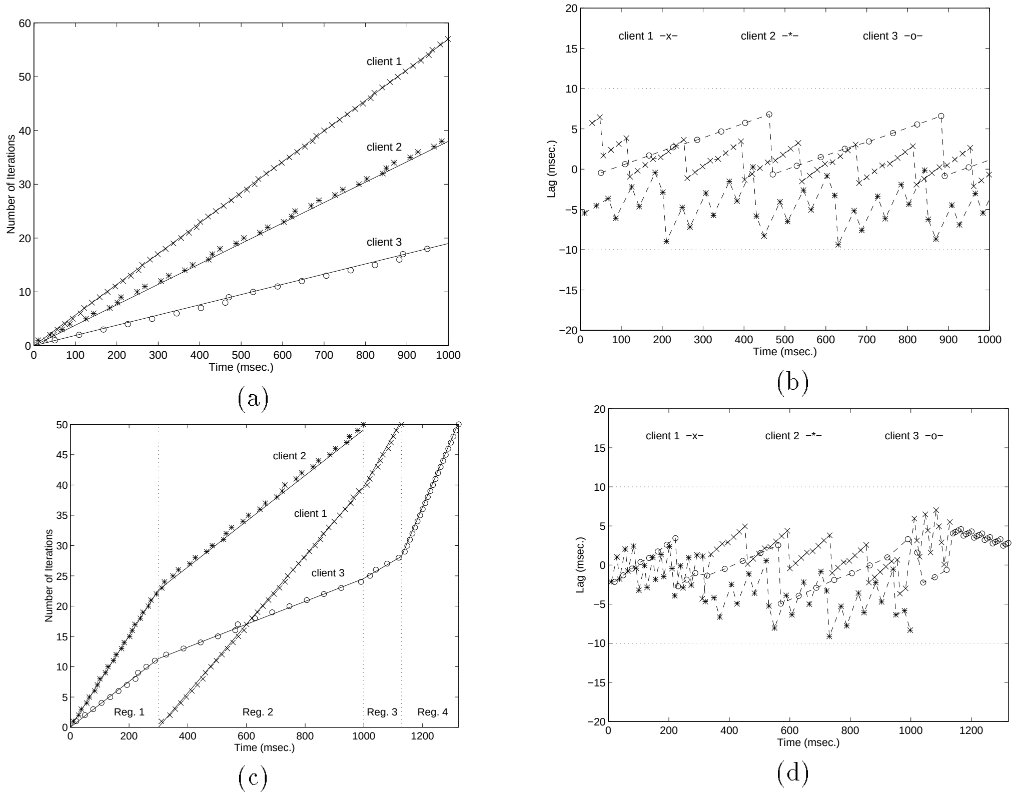

Figure 3. *Experiment 1: The number of iterations (a) and lags (b) of three clients with weights 3, 2, and 1 over 1 sec period. Experiment 2: The number of iterations (c) and the lags (d) for the same clients, when client 3 is delayed for 300 ms and each client performs 50 iterations.*

by our scheduler. All the other processes are scheduled by the regular FreeBSD scheduler. In this way, the kernel processes are scheduled over any process in the proportional share or the reservation class. Since FreeBSD lacks real-time support, such as preemptive kernel or page pinning, in our experiments we tried to avoid as much as possible the interaction with the kernel. For example, we make sure that all the measurements are performed after the entire program is loaded into memory Also, with the exception of `gettimeofday` function used for time measurements, we do not use any other system function while running the experiments (all the data are recorded in memory and saved at the end of each experiment). Addressing these issues in a first-class manner would only serve to improve our (already good) results.

To measure the allocation accuracy we have written a simple iterative applications which performs some arithmetic computations. Each iteration takes close to 9 ms. In each experiment we run several copies of the program, by assigning to each copy (client) a different weight.

In the first experiment we run three clients (processes), with weights 3, 2, and 1, respectively. All clients are synchronized via shared-memory to start the actual computation at the same time. Figure 3(a) shows the number of iterations executed by each client during the first second. The solid lines depict the ideal number of iterations for each client. As it can be seen, the number of iterations actually performed by each client is very closed to the ideal one. In particular, note that the client with a weight of 1 executes 1/2 the number of iterations as the client with a weight of 2 and at 1/3 the rate as the client with weight 3. Figure 3(b) depicts the lag of each client over the same interval. Note that the lags are always between -10 and 10 ms, which is consistent with the bounds given by the corollary in Section 6. Thus, each client is executing at a rate which is precise enough to afford one the ability to predict its performance in real-time (modulo 10 ms.) over any interval.

In the second experiment we consider again three clients with weights 3, 2, and 1, respectively, but in a more "dynamic" scenario. While clients 2 and 3 begin execution at the same time, client 1 is delayed for 300 ms. Each client performs 50 iterations after which it leaves the competition. As shown in Figure 3(c) there are four distinct regions. In the first region (i.e., between 0 and 300 ms) there are only two active clients: 2 and 3. Therefore client 2 (having weight 2) receives 66% percent of the CPU, while client 3 (having weight 1) receives 33% from the CPU. Consequently, after 300 ms client 2 completes 22 iterations, while client 3 completes only 11 iterations. After 300 ms, client 1 joins the competition and therefore in the second region (between 300 and 998 ms) all three clients are active. Further, at time $t = 998$ ms client 2 finishes all its iterations and leaves the competition. Thus, in the next region only clients 1 and 3 remains active. Finally, at time $t = 1128$ ms, client 1 finishes, and client 3 remain the only one active (in region four). Figure 3(d) depicts the clients lags, which are again between the theoretical bounds, i.e, -10 and 10 ms.

8 Conclusions

We have described a new proportional share resource allocation scheduler that provides a flexible control, and provides strong timeliness guarantees for the service time received by a client. In this way we provide a unified approach for scheduling "firm" real-time, interactive, and batch applications. We achieve this by uniformly converting the application requirements regardless of their type in a sequence of requests for the resource. Our algorithm guarantees that the difference between the service time that a client should receive in the idealized system and the service time it actually receives in the real system is bounded by one time quantum and that this bound is optimal. At our best knowledge, this is the first algorithm to achieve these bounds in a dynamic system that provides support for both fractional and non-uniform quanta. As a proof of concept we have also implemented a prototype of a CPU scheduler under the FreeBSD operating system. Our experimental results shows that our implementation performs within the theoretical bounds.

References

[1] S. K. Baruah, J. E. Gehrke and C. G. Plaxton, "Fast Scheduling of Periodic Tasks on Multiple Resources", *Proc. of the 9th Int. Par. Proc. Symp.*, April 1995, pp. 280–288.

[2] S. K. Baruah, J. E. Gehrke and C. G. Plaxton, "Fair On-Line Scheduling of a Dynamic Set of Tasks on a Single Resource", *Technical Report* UTCS-TR-96-03, Dpt. of CS, Univ. of Texas at Austin, February 1996.

[3] J. C. R. Bennett and H. Zhang, "WF^{2}Q : Worst-case Fair Queueing", *Proc. of INFOCOM'96*, San-Francisco, March 1996.

[4] A. Demers, S. Keshav and S. Shenkar, "Analysis and Simulation of a Fair Queueing Algorithm", *Journal of Internetworking Research & Experience*, October 1990, pp. 3–12.

[5] S. J. Golestani, "A Self-Clocked Fair Queueing Scheme for Broadband Applications", *Proc. of INFOCOM'94*, April 1994, pp. 636–646.

[6] P. Goyal, X. Guo and H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems", to appear in *Proc. of the 2nd OSDI Symp.*, October 1996.

[7] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman. "The Design and Implementation of the 4.3BSD UNIX Operating System," Addison-Wesley, 1989.

[8] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, Vol. 20, No. 1, January 1973, pp. 46–61.

[9] U. Maheshwari, "Charged-based Proportional Scheduling", Technical Memorandum MIT/LCS/TM-529, Laboratory for CS, MIT, July 1995.

[10] A. K. Parekh and R. G. Gallager, "A Generalized Processor Sharing Approach To Flow Control in Integrated Services Networks-The Single Node Case", *ACM/IEEE Trans. on Networking*, Vol. 1, No. 3, 1992, pp. 344–357.

[11] A. K. Parekh, "A Generalized Processor Sharing Approach To Flow Control in Integrated Services Networks", *Ph.D Thesis*, Department of EE and CS, MIT, 1992.

[12] I. Stoica, H. Abdel-Wahab, "A new approach to implement proportional share resource allocation", *Technical Report* TR-95-05, CS Dpt., Old Dominion Univ., April 1995.

[13] I. Stoica, H. Abdel-Wahab, "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation", *Technical Report* TR-95-22, CS Dpt., Old Dominion Univ., Nov. 1995.

[14] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke and C. G. Plaxton, "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems", *Technical Report* TR-96-38, CS Dpt., Univ. of North Carolina, September 1996.

[15] R. Tijdeman, "The Chairmain Assignment Problem", *Discrete Mathematics*, vol. 32, 1980, pp. 323–330.

[16] C. A. Waldspurger and W. E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. of the 1st OSDI Symp.*, November 1994, pp. 1–12.

[17] C. A. Waldspurger and W. E. Weihl. "Stride Scheduling: Deterministic Proportional Share Resource Menagement," Technical Memorandum, MIT/LCS/TM-528, Laboratory for CS, MIT, July 1995.

[18] C. A. Waldspurger. "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," *PhD Thesis*, Technical Report, MIT/LCS/TR-667, Laboratory for CS, MIT, September 1995.

[19] L. Zhang, "VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks", *ACM Trans. on Comp. Systems*, vol. 9, no. 2, May 1991, pp. 101–124.