Multiprocessor Scheduling in Processor-based Router Platforms: *Issues and Ideas* *

Anand Srinivasan, Philip Holman, James Anderson, Sanjoy Baruah, and Jasleen Kaur
Department of Computer Science, University of North Carolina at Chapel Hill
E-mail: {anands,holman,anderson,baruah,jasleen}@cs.unc.edu

Abstract

Two important trends are expected to guide the design of next-generation networks. First, with the commercialization of the Internet, providers will use value-added services to differentiate their service offerings from other providers; many services require the use of sophisticated resource scheduling mechanisms in routers. Second, to enable extensibility and the deployment of new services in a rapid and cost-effective manner, routers will be instantiated using programmable network processors. Our research objective is to develop sophisticated multiprocessor scheduling mechanisms that would enable networks that deploy such router platforms to provide service guarantees to applications. In this paper, we make two contributions. First, we identify the issues that arise when existing multiprocessor scheduling techniques are used in routers. We find that existing techniques are either not applicable to router platforms due to their complexity or simplistic assumptions, or are not based on rigorous formalism, which is necessary to enable strong assertions about service guarantees. Second, we present key ideas (and planned research directions) that we believe will enable us to address these limitations.

1 Introduction

With the commercialization of the Internet, there is a growing need for network providers to enhance their service offerings. Deployment of new services often requires making changes to switches and routers, which are the basic building blocks of wide-area networks such as the Internet. Traditionally, routers have been built using application-specific integrated circuits (ASICs) that enable high-speed packet switching. Unfortunately, ASIC designs take months to develop, and routers built using them are costly to deploy. In order to enable router extensibility in a rapid and cost-effective manner, significant effort is now being invested in a different approach: implementing routers using programmable network processors (NPs) [1, 2, 3, 33].

There are three main shared resources in a software-based programmable router: *link capacity*, which is shared by traffic destined for the same outgoing link, *memory capacity* and *packet-processing capacity*,[1] which are shared by all traffic arriving at a router. Two trends are expected to guide the manner in which these resources are managed in next-generation routers:

- **Growing demand for sophisticated resource-allocation mechanisms.** The current Internet mainly supports just a single service class, namely, best-effort delivery. In this model, there is no assurance of when, or even if, a packet sent by a data source will reach its destination. While this model has worked well for traditional applications such as email and web browsing, it is not adequate for many emerging network applications that require quality-of-service and timeliness guarantees. Such applications require that both link and packet-processing capacities be mul-

*Work supported by NSF grants CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0204312.

[1] The term *packet processing* refers to functions that are performed for every incoming packet, such as computing checksums, route lookup, packet classification, *etc.*

tiplexed across different applications in a *fair* manner, even at short time scales.

- **Increasing disparity between link and processor speeds.** Link capacities are increasing rapidly, almost doubling every year [16]. On the other hand, processor speeds are increasing less rapidly [4]. This gap is further heightened by the fact that the processing demands of new applications — such as encryption/decryption or transcoding — are increasing. For these reasons, routers must be instantiated as *multiprocessor platforms* that process multiple packets simultaneously. Further, to keep up with large volumes of Internet traffic [16, 26], router mechanisms must incur *very low overhead*.

In the last decade, a significant amount of research has been conducted on scheduling mechanisms for fairly allocating link capacity [11, 12, 18, 19, 20]. However, considerably less work has been done on designing mechanisms for allocating packet-processing capacity. The reason for this is simple: conventional routers, built using ASICs, perform only simple packet-processing functions that are likely to execute faster than the time it takes to transmit packets between ports. As such, link capacities are assumed to be the only resources in a network for which flows must contend. However, routers built using programmable NPs are destined to implement more complex packet-processing functions in software, making packet-processing capacity a critical resource to be managed.

Unfortunately, techniques developed in prior work on link scheduling cannot be directly applied to the problem of fairly allocating packet-processing capacity in multiprocessor routers. There are two reasons for this. First, link-scheduling algorithms are typically devised to manage just a single outgoing link, *i.e.*, link scheduling is fundamentally a *single*-resource scheduling problem. Second, some assumptions usually made in work on link scheduling — *e.g.*, unbounded buffering capacity and processing bandwidth — are unreasonable to assume on router platforms connected to high-speed links.

Given the trends noted above, and the limitations of prior work, there is a significant need for research on the problem of fairly allocating packet-processing capacity in multiprocessor routers. Fair scheduling on multiprocessors has been a topic of recent research in work on real-time scheduling, and several fair scheduling algorithms have been developed [6, 9, 10, 22, 36]. However, due to various assumptions made in this work, these algorithms cannot be applied directly in multiprocessor routers. In this paper, we explore some of the issues that arise in applying these algorithms to routers, and describe some approaches to handle them.

The rest of this paper is organized as follows. In Sec. 2, we consider the problem of scheduling packet-processing capacity in routers in more detail and also describe past work on multiprocessor scheduling in real-time systems. In Sec. 3, we consider requirements unique to router platforms and outline some of the issues in applying existing multiprocessor scheduling schemes to routers. In Sec. 4, we present some ideas for addressing these issues. In Sec. 5, we present an experimental methodology to evaluate our approach. Finally, we summarize in Sec. 6.

2 Related Work and Concepts

In this section, we formulate the problem of multiprocessor scheduling in routers and describe prior work on multiprocessor scheduling.

2.1 The Problem: Limited Packet-processing Capacity in Routers

Fig. 1(a) depicts the high-level architecture of a typical wide-area packet-switched network. When a packet arrives on an input link at a router, the router determines the *next* hop on the end-to-end path of that packet, and transmits the packet on the corresponding output link.

To understand the need for scheduling mechanisms in routers, consider Fig. 1(b), which depicts the architecture of a typical router platform

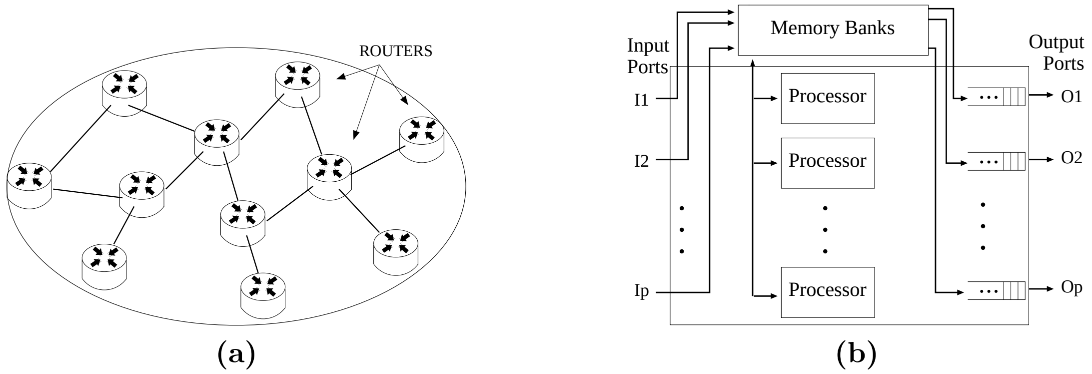

Figure 1: Typical architectures of **(a)** a network, and **(b)** a programmable NP-based router.

built using programmable NPs.[2] Each incoming packet is stored in memory, is processed at one or more of the M processors, and is transmitted on an outgoing link. The processing required for each packet includes such functions as checksum computation, packet classification, route-table lookup, queue maintenance, *etc.*

As mentioned earlier, there are three main resources for which the packets of different flows contend: **(i)** processing capacity within the multiprocessor bank, **(ii)** transmission capacity of the outgoing links, and **(ii)** buffer space in the high-speed memory banks. In order to provide delay and throughput guarantees for the packets of a given flow, the router must employ sophisticated scheduling mechanisms to arbitrate access to the first two resources, and mechanisms to allocate, in a fair manner, buffer space. In this paper, we focus only on the problem of allocating processing capacity.

Research on designing scheduling mechanisms for routers over the past decade has focused mainly on the second resource identified above, *i.e.*, outgoing link capacity [12, 18, 19, 20, 32, 38]. Indeed, in most work on link-scheduling algorithms, buffer space and processing capacity within routers are assumed to be unlimited. These assumptions have been justified by the fact that conventional routers, built using ASICs, have very fast interconnects between input and output ports, and perform simple packet-processing functions implemented using fast hardware. Consequently, it is reasonable to assume that packets require queueing only while accessing outgoing links. However, because of the two trends mentioned earlier — the growing disparity between link and processor speeds, and an increasing need for more sophisticated resource-allocation mechanisms — it is not reasonable to assume that queueing occurs only at outgoing links in software-based router platforms built using NPs. In this paper, we focus on the problem of fairly allocating processing capacity in multiprocessor routers.

[2]The architecture shown here, in which streams arriving on multiple input links are processed in the same NP, is based on that of Intel's IXP1200 [1]. Some NPs are used in line cards attached to a single link. In such architectures, the streams shown in Fig. 1(b) arrive and depart on a single link.

2.2 Real-time Scheduling on Multiprocessors

The problem of fairly scheduling tasks on multiprocessors has recently received much attention in the real-time scheduling literature [6, 8, 22, 31, 36]. Task models such as the periodic and sporadic models, which permit *recurrent* execution, are central to the theory of real-time scheduling. In the *periodic* model, each task is invoked repeatedly, with consecutive invocations, or *jobs*, spaced apart by a fixed amount, called the task's *period*. In the *sporadic* task model, job deadlines are defined similarly, but a task's period defines a *lower bound* between successive jobs. In the variant of these models considered here, each task's *relative job deadline* is equal to its period: each job of a task must complete execution before the current period of that task has elapsed. In both models, the *weight* or *utilization* of a task T, denoted $wt(T)$, is defined as the ratio of its per-job execution requirement and its period.

Multiprocessor scheduling techniques in real-time systems fall into two general categories: *partitioning* and *global scheduling*. Under partitioning, each processor schedules tasks independently from a local ready queue. Each task is assigned to a particular processor and is only scheduled on that processor. In contrast, all ready tasks are stored in a single queue under global scheduling. A single system-wide priority space is assumed; the highest-priority task is selected to execute whenever the scheduler is invoked, regardless of which processor is being scheduled.

Partitioning has two main advantages: migration overhead is zero (since each task runs on only one processor) and simpler and widely-studied uniprocessor scheduling algorithms can be used on each processor. Unfortunately, finding an optimal assignment of tasks to processors is a bin-packing problem, which is NP-hard in the strong sense. In addition, partitioning is inherently sub-optimal: task sets with utilization at most M exist that cannot be partitioned on M processors. Furthermore, partitioning is quite problematic if tasks can be created and terminated dynamically. In particular, every new task that joins can potentially cause a re-partitioning of the entire system.

Because of the limitations of partitioning, there has been much recent interest in global multiprocessor scheduling algorithms that ensure *fairness* [8, 13, 14, 22, 36]. In the following subsection, we present an overview of this work.

2.3 Pfair Scheduling

Fairness is defined with respect to a basic *fluid-flow model*. Given a resource that is shared among several data streams (or tasks) in specified proportions, the goal of a fair scheduling algorithm is to allocate the resource so that each stream's allocation is always "close" to its designated proportion. Ideally, one would like to treat the data streams as fluid flows and continually assign an appropriate fraction of the available bandwidth to each stream. This idealized scheme is referred to as *generalized processor sharing* (GPS) [32]. In practice, supporting ideal fluid flows is not possible, and hence scheduling schemes that closely approximate GPS must be used. Fairness in such schemes is usually measured by determining *lag bounds* that reflect the extent to which actual behavior deviates from the idealized GPS behavior.

Proportionate fairness (Pfairness) is a fairness constraint introduced by Baruah *et al.* [9, 10] as a way to optimally schedule periodic tasks on multiprocessors. Some of the key concepts arising in this work are described below.

Assumptions and terminology. Most prior work on Pfairness has focused on periodic tasks with hard real-time execution requirements, where processor time is allocated in uniform time units called *time slots* or *quanta*. Let $T.p$ denote the period of task T, and let $T.e$ denote its execution requirement within each of its periods. Then, $wt(T) = T.e/T.p$ is the processing-rate requirement of T. In satisfying this rate requirement, T may be allocated time on different processors, but not at the same time (*i.e.*, migration is allowed but parallelism is not).

Pfairness. Pfairness is defined by focusing on the *lag* between the amount of time allocated to each task and the amount of time that would be allocated to that task in an ideal fluid system (*i.e.*, GPS). Formally, the *lag of task T at time t*, denoted $lag(T, t)$, is defined as follows:

$$lag(T, t) = wt(T) \cdot t - allocated(T, t),$$

where $allocated(T, t)$ is the total processor time allocated to T in $[0, t)$. A schedule is *Pfair*[3] iff

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

Informally, the allocation error associated with each task must always be strictly less than one quantum. It is easy to show that (1) ensures that each job completes before the next job of the same task is released.

[3] A lower bound of 0 for $lag(T, t)$ (as in most uniprocessor fair scheduling schemes) is not sufficient to guarantee all deadlines on multiprocessors.

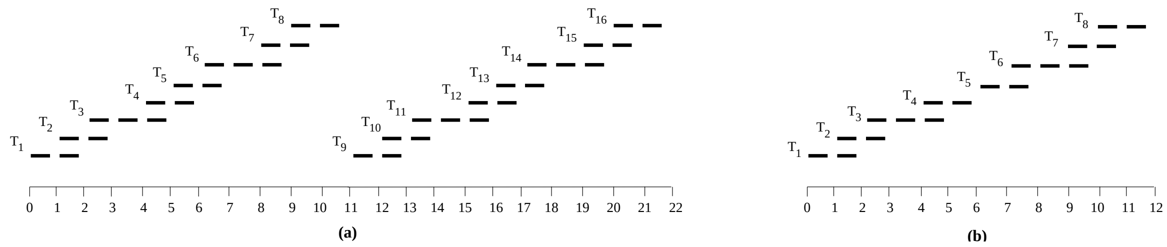

Figure 2: **(a)** Windows of the first two jobs of a periodic task T with weight 8/11. These two jobs consist of the subtasks $T_1, \ldots, T_8$ and $T_9, \ldots, T_{16}$, respectively. Each subtask must be scheduled during its window, or a lag-bound violation will result. **(b)** The Pfair windows of an IS task. Subtask T_5 becomes eligible one unit late.

Baruah *et al.* [9] proved that a periodic task system τ has a Pfair schedule on M processors iff

$$\sum_{T\in\tau} wt(T) \leq M.$$

This expression is in fact a feasibility condition for all the task models considered in this subsection.

Pfair scheduling algorithms. Baruah *et al.* presented two optimal Pfair scheduling algorithms, called PF [9] and PD [10]. In these algorithms, a task T is divided into a sequence $T_1, T_2, \ldots$ of quantum-length *subtasks* to be executed sequentially. Both algorithms schedule subtasks on an *earliest deadline first* basis, where the deadline of a subtask T_i ($i \geq 1$), denoted $d(T_i)$, is computed as follows.

$$d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \tag{2}$$

These deadline assignments respect the lag bounds given in (1). The lag bounds in (1) also imply a *release* time for a subtask, *i.e.*, the earliest time at which the subtask can be scheduled. Thus, we obtain a time interval for each subtask during which that subtask must be scheduled. This interval is referred to as the subtask's *window* (see Fig. 2(a)).

PF and PD differ in the way in which ties are broken when two subtasks have the same deadline. (Selecting appropriate tie-breaks turns out to be *the* most important concern in designing correct Pfair algorithms.) In PF, ties are broken by comparing a vector of future subtask deadlines, which is somewhat expensive. In PD, ties are broken in constant time by inspecting four tie-break parameters. In recent work, Anderson and Srinivasan presented an optimized variant of PD, called PD^2 [6, 8], which was obtained by eliminating two of PD's tie-breaks. PD^2 is the most efficient Pfair scheduling algorithm currently known. We now examine the PD^2 tie-breaks in more detail.

PD^2 tie-breaks. The first PD^2 tie-break is a bit, denoted $b(T_i)$. As seen in Fig. 2(a), consecutive windows of a Pfair task are either disjoint or overlap by one slot. $b(T_i)$ is defined to be 1 if T_i's window overlaps T_{i+1}'s, and 0 otherwise. For example, for task T in Fig. 2(a), $b(T_i) = 1$ for $1 \leq i \leq 7$ and $b(T_8) = 0$. PD^2 favors a subtask with a b-bit of 1 over one with a b-bit of 0. Informally, it is better to execute T_i "early" if its window overlaps that of T_{i+1}, because this potentially leaves more slots available to T_{i+1}.

The second PD^2 tie-break, the *group deadline*, is needed in systems containing tasks with windows of length two. A task T has such windows iff $1/2 \leq wt(T) < 1$. Consider a sequence $T_i, \ldots, T_j$ of subtasks of such a task T such that $b(T_k) = 1 \ \wedge \ |w(T_{k+1})| = 2$ for all $i \leq k < j$. Scheduling T_i in its last slot forces the other subtasks in this sequence to be scheduled in their last slots. For example, in Fig. 2(a), scheduling T_3 in slot 4 forces T_4 and T_5 to be scheduled in slots 5 and 6, respectively. The group deadline of a subtask T_i is the earliest time by which such a "cascade" must end. Formally, it is the earliest time t, where $t \geq d(T_i)$, such that either $(t = d(T_k) \wedge b(T_k) = 0)$ or $(t + 1 = d(T_k) \ \wedge \ |w(T_k)| = 3)$ for some subtask T_k. For example, subtask T_3 in Fig. 2(a) has a group deadline at time 8 and subtask T_7 has a group deadline at time 11. PD^2 favors subtasks with later group deadlines because *not* scheduling them can lead to longer cascades.

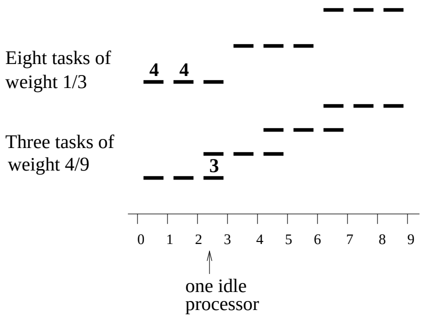

Figure 3: Eight tasks of weight 1/3 and four tasks of weight 4/9 scheduled on four processors (tasks of a given weight are shown together). The Pfair window of each subtask is shown on a separate line. An integer value n in slot t means that n of the corresponding subtasks are scheduled in slot t.

Anderson and Srinivasan have shown that if either PD^2 tie-break is eliminated, then tasks can miss their deadlines. To see that the b-bit is necessary, consider Fig. 3. In this schedule, the tasks of weight 1/3 are favored over those of weight 4/9 at times 0 and 1 even though the former have a b-bit of 0. Note that $\frac{8}{3} + \frac{4}{3} = 4$. Thus, all four processors are fully utilized, which implies that no processor should ever be idle. However, in $[2, 3)$, only three tasks can be scheduled, implying that a deadline is missed in the future.

Allowing early releases and late arrivals. Pfair scheduling algorithms are necessarily *non-work conserving* when used to schedule periodic tasks. To see why, suppose some subtask T_i executes "early" within its window. Then T_{i+1}, the next subtask of T, will be ineligible for execution until the beginning of its window, even if some processors are idle. To enable more efficient use of processing capacity, a work-conserving variant of Pfair scheduling called *"early-release" fair* (ERfair) scheduling was recently proposed by Anderson and Srinivasan [6, 8]. Under ERfair scheduling, if two subtasks are part of the same job, then the second subtask becomes eligible for execution as soon as the first completes. For example, if T_3 in Fig. 2(a) were scheduled in slot 2, then T_4 could be scheduled as early as slot 3.

In other recent work, Anderson and Srinivasan extended the early-release task model to also allow subtasks to be released "late," *i.e.*, there may be separation between consecutive subtasks of the same task [7]. The resulting model, called the *intra-sporadic* (IS) model, generalizes the sporadic model, which allows separation between consecutive *jobs* of the same task. An example of an IS task is shown in Fig. 2(b), where T_5 is released one slot late. Note that an IS task is obtained by allowing a task's windows to be right-shifted from where they would appear if the task were periodic. Thus, we can define an IS task by associating with each subtask an *offset* that gives the amount by which its window has been right-shifted. Let $\theta(T_i)$ denote the offset of subtask T_i. Then, from (2), we have the following.

$$d(T_i) = \theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil \qquad (3)$$

These offsets are constrained so that the separation between any pair of subtask deadlines is no less than the separation between those deadlines if the task were periodic. Formally, the offsets satisfy the following: $k \geq i \Rightarrow \theta(T_k) \geq \theta(T_i)$. Anderson and Srinivasan have shown that IS (and hence early-release) tasks can be correctly scheduled by PD^2 on M processors if total utilization is at most M [36].

Heuristic approaches. In recent work, Chandra *et al.* considered fair multiprocessor scheduling algorithms that use variable-sized quanta, use mechanisms that discourage task migrations, and allow tasks to join and leave dynamically [13, 14]. Their work is entirely experimental in nature. In particular, they provide no formal correctness proofs for the algorithms they consider. Nonetheless, their results demonstrate convincingly the utility of using fair scheduling algorithms on multiprocessors.

In other recent work, Jones and Regehr proposed and evaluated a reservation-based multiprocessor scheduler implemented within a research version of Windows NT called Rialto/NT [25]. While their results show that reservations and real-time execution can be effectively implemented on multiprocessors, Jones and Regehr, like Chandra *et al.*, present no formal analysis of their scheduling algorithm.

3 Issues in Using Pfair Schedulers in Routers

We now examine how real-time task scheduling concepts can be applied to the problem of packet processing on multiprocessor routers. In Pfair terms, the arrival of a packet for a given connection would cause a task (corresponding to that connection) to release one or more subtasks that encompass the packet-processing functions to be performed.

While the notion of a Pfair weight was defined above based on the per-job execution cost and period of a task, these weights can be viewed more abstractly as denoting *maximum processor shares*. In packet scheduling, each backlogged flow must instead be guaranteed a *minimum* share. If the workload to be scheduled never changes, then the share of each flow remains fixed and there is no real distinction between the notion of a maximum and a minimum share. However, in a dynamic system, in which flows may join and leave or become inactive, it is desirable to increase a task's share if there is available spare capacity. In fact, this very issue was one of the key problems addressed in prior work on fair link scheduling.

Of the task models considered above, the intra-sporadic (IS) model is the most suitable for scheduling dynamic flows within a router. Due to network congestion and other factors, packet arrivals may not be periodic, and packets may arrive late or in bursts. The IS model treats these possibilities as first-class concepts. In particular, a late packet arrival corresponds to an IS delay. On the other hand, if a packet arrives early (as part of a bursty sequence), then eligibility times of the corresponding subtasks will be less than their Pfair release times. Note that the Pfair release time of a subtask determines its deadline. Thus, in effect, an early packet arrival is handled by postponing the deadline of the subtasks to where they would have been had the packet arrived on time. This is very similar to the approach taken in the (uniprocessor) virtual-clock scheduling scheme [39].

The observations made above suggest that Pfair scheduling schemes that allow IS task execution are capable of providing provable service guarantees, while being an efficient and flexible choice for multiprocessor router platforms. We now examine this choice in greater detail. Below, we describe several issues that indicate that existing Pfair scheduling algorithms need to be refined for use in multiprocessor routers.

3.1 Applicability Issues

Some assumptions made in Pfair scheduling theory are not valid in the context of packet processing in routers. Specifically, in all prior work on Pfair algorithms, quanta have been assumed to be uniform, and to always align on all processors (see Fig. 2). However, the execution costs of packet-processing functions in routers can vary significantly. In particular, packets (even from the same flow) may vary in size, and the complexity of some processing functions (*e.g.*, validating the checksum of a packet) is a function of packet size. Furthermore, if different threads are responsible for different subsets of processing functions, the execution cost across threads may vary substantially. Hence, if a fixed-length quantum is used, then some quanta will almost surely be partially wasted. Specifically, waste will occur whenever packet processing functions complete before the next quantum boundary.[4] In addition, with aligned quanta, there may be excessive memory contention at the beginning of each quantum. Thus, to enable efficient use of resources, *multiprocessor scheduling schemes used on routers should allow non-uniform and non-aligned quanta*.

A second key assumption in prior work on Pfair scheduling is that task migrations are unconstrained. However, on a router, such migrations may need to be constrained to reduce the number of (off-chip) memory references made per packet.

[4] Note that it is possible to reduce this loss by scheduling a new thread for the rest of the quantum. However, the loss that we are considering here is the loss due to inflating execution costs to a multiple of the quantum size. For instance, if $T.e = 1.5$ and $T.p = 3$, then $wt(T)$ (which determines the utilization reserved for task T) would be defined as $2/3$ instead of $1/2$. This reduces the total number of tasks that will be accepted by the scheduler, and hence, reduces the utilization of the system.

Migrations may also be constrained by the processing architecture. For instance, if different processing functions are pipelined, then the task corresponding to the processing of a particular packet may necessarily migrate in a constrained (not arbitrary) pattern. Therefore, current multiprocessor scheduling theory needs to be extended to *operate under constrained models of migration.*

3.2 Flexibility Issues

Unlike many real-time applications built for stand-alone embedded systems, packet arrivals for a given flow at a router are not likely to be periodic. Packets may arrive at a rate less than or greater than the rate reserved by the flow, and may end up creating or utilizing spare processing capacity.

In contrast, even in the most flexible task model used for Pfair scheduling, namely the IS model, subtask deadlines are computed according to a strictly periodic schedule of packet arrivals. Scheduling algorithms designed specifically for periodic flows may penalize flows that use spare capacity to transmit at more than their reserved rates, by denying them allocation at their reserved rate during a subsequent time interval. To see this, consider again Fig. 2(a). Suppose that early releases are allowed, and there is spare capacity prior to time 11, but not afterwards. Then, T could potentially execute in each of slots 0 through 10. These eleven subtasks "use up" the first eleven subtask deadlines of T. Thus, if T has another eligible subtask at time 11, then it uses the subtask deadline associated with T_{12}, which is at time 17. Thus, while each subtask of T *should* have a deadline two or three time units after its release, this particular subtask has a deadline 6 time units after its release. In general, the extent to which deadlines can be postponed in this manner is unbounded.

This property of penalizing flows for using spare capacity is undesirable in networks for two reasons. First, for many network applications that have timeliness requirements, it may not be feasible to predict accurately the exact rate to reserve. (Consider, for instance, the problem of transmitting a variable bit-rate encoded video stream over the network.) The performance of such an application may be significantly enhanced by allowing it to utilize spare capacity. Second, allowing applications to transmit packets in bursts enables networks to provide low average delays and to increase network utilization due to statistical multiplexing gains. Thus, it is important to *devise fair allocation schemes for multiprocessors that do not penalize flows for using spare processing capacity in the past.*

3.3 Performance Issues

Network link capacities are increasing at almost double the rate at which router processing speeds are increasing [4, 16]. To keep up with high link speeds, packet-processing functions must be implemented in a highly-efficient manner. Deadline-based scheduling algorithms, on the other hand, impose non-negligible computational complexity. In particular, these algorithms require routers to maintain a sorted queue of packet descriptors (sorted according to deadlines); this leads to a per-packet complexity that is a function of the number of packets (or flows). Therefore, to enable routers to operate efficiently in high-speed networks, *the complexity associated with packet sorting should be eliminated, while ensuring that meaningful deadline guarantees are still provided.*

3.4 Scalability Issues

To compute deadlines, routers need to maintain state (for instance, $\theta(T_i)$ in Equation (3)) for each active flow and perform flow classification for all incoming packets. However, the complexity of these two per-flow operations limits the scalability of routers as the number of flows increases. This is especially true for routers in the core of a network, which aggregate and handle a large number of flows originating from different edges of the network, and are also required to operate on high-speed links. Thus, it is essential to *eliminate the complexity associated with these per-flow operations in the core routers of a network.*

4 Multiprocessor Scheduling in Routers: *Key Ideas*

In this section, we discuss some approaches that we believe will address the issues outlined in Sec. 3. Many of the ideas presented in this section exploit a key difference between the timeliness requirements of traditional real-time applications and emerging network applications. Network applications have to be designed to tolerate the least possible network delay, which is given by the sum of link-propagation and transmission latencies on end-to-end paths. These minimum end-to-end latencies are themselves of the order of a few *milliseconds*. Unlike hard real-time applications that have no tolerance for deadline misses, these applications are capable of operating well even if deadlines are not strictly adhered to, as long as deadline misses are bounded by a quantity less than, say, a fraction of a millisecond. Hence, the use of PD^2 is not warranted and a simpler algorithm may be used.

The EPDF Pfair algorithm is similar to PD^2, but *uses no tie-breaks*. While deadline misses can occur under EPDF (refer to Fig. 3), based upon preliminary evidence presented below, we believe that the impact of such misses will be extremely limited.

Deadline guarantees under EPDF. The *tardiness* of a scheduling algorithm indicates the extent to which a deadline can be missed. To determine how frequently deadlines are missed under EPDF, and by what tardiness threshold, we recently constructed a series of EPDF schedules for randomly-generated task sets [35]. Out of approximately 200,000 generated task sets, *no subtask ever missed its deadline by more than one quantum*. Moreover, single-quantum deadline misses were very rare; *e.g.*, on systems of five or more processors, the miss rate was about 0.1%. This evidence indicates that EPDF (and variants considered in Sec. 4.1) may be ideal for use within multiprocessor routers.

Currently, we are trying to establish *formally* that EPDF guarantees a tardiness of one. To date, this has been proved for systems that satisfy the following constraint [35].

(M1) The sum of the weights of the $M-1$ heaviest tasks is at most $(M+1)/(2M-3)$.

Note that this restriction only applies if there are tasks with weight greater than $1/2$ and imposes no restrictions on systems of four or fewer processors. It may seem that eliminating such a liberal condition is not important. After all, a flow with weight exceeding $1/2$ may seem quite unlikely. However, one of the scheduling schemes considered later in Sec. 4.1.2 is a hierarchical scheme in which several tasks are bundled together into a single "supertask." Such a supertask can easily have a weight exceeding $1/2$. Furthermore, many of the scheduling problems considered in Sec. 4.1 also involve establishing tardiness thresholds. We regard the problem of establishing EPDF tardiness as an important "bellwether" problem in this class.

4.1 Applicability Issues

We now present ideas for addressing the two assumptions of *uniform and aligned quanta* and *unconstrained migration*, which limit the applicability of Pfair scheduling algorithms for packet processing in routers.

4.1.1 Allowing Non-uniform and Non-aligned Quanta

Fixed-length quanta. Allowing only fixed-length quanta simplifies the analysis of multiprocessor schedulers; this, however, may result in lower utilization due to partially-wasted quanta. An obvious way to reduce the extent of waste is to use a smaller quantum size. Unfortunately, scheduling at a finer granularity in this way increases context-switching and scheduling overheads, and thereby reduces the amount of processor time available for tasks. Thus, there is a trade-off between system overheads and partially-wasted quanta that needs to be carefully analyzed to determine an optimal quantum size.

Interconnect contention due to aligned quanta could possibly be avoided by staggering quantum allocations on each processor as shown in Fig. 4.

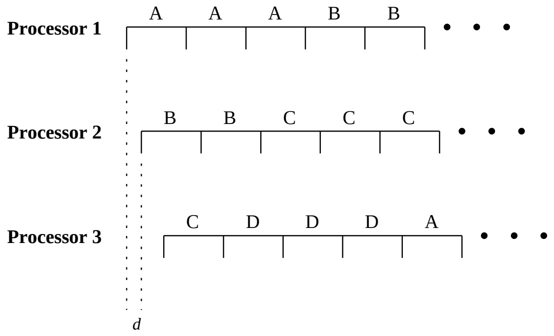

Figure 4: A partial schedule on three processors with staggered quantum allocations. This schedule is shown differently from those in other figures: each line shows quantum allocations on one processor. Four tasks, denoted A through D, of weight $3/4$ each are scheduled. The t^{th} quantum ($t \geq 0$) begins at time $t + d(i-1)$ on processor i. (Note that it may not be always possible to migrate a task that is scheduled in two consecutive slots. Thus, we have additional restrictions.)

The exact extent to which such staggering might impact the algorithms and techniques described in this paper remains to be determined.

Variable-length quanta. A liberal quantum-allocation model is one in which quanta may vary in length (up to some threshold) and do not have to align. While allowing variable-length quanta would probably be disastrous if task deadlines were hard, we are somewhat optimistic that the impact may be acceptable if deadlines can be missed. We are currently analyzing EPDF for use in this model and trying to derive corresponding tardiness thresholds.

4.1.2 Limiting Task Migrations

For systems with non-migratory tasks, the hierarchical *supertask* approach proposed by Moir and Ramamurthy [31] can be used. In this approach, the non-migratory tasks bound to a specific processor are combined into a single "supertask," which is then scheduled as an ordinary Pfair task; when a supertask is scheduled, one of its component non-migratory tasks is selected for execution. Unfortunately, while supertasking is a promising approach, the following example illustrates that non-migratory tasks can actually miss their deadlines when supertasking is used in conjunction with PF, PD, or PD^2.

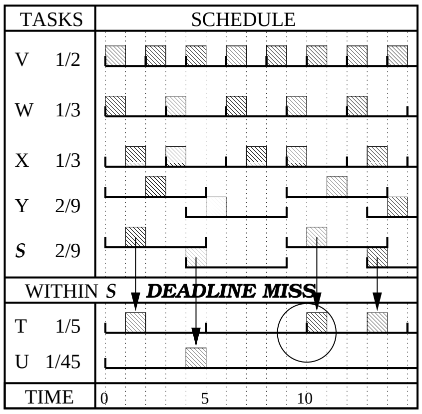

Figure 5: PD^2/EPDF schedule with a supertask S on two processors.

Example *To see why supertasking can fail, consider the two-processor Pfair schedule shown in Fig. 5. In this schedule, there are four normal tasks V, W, X, and Y with weights $1/2$, $1/3$, $1/3$, and $2/9$, respectively, and one supertask, S, which represents two component tasks T and U, with weights $1/5$ and $1/45$, respectively. S competes with a weight of $1/5 + 1/45 = 2/9$. The windows of each task are shown on alternating lines (e.g., S's first window spans $[0, 5)$), and a shaded box denotes the quantum allocated to each subtask. In the upper region of the figure, the PD^2 schedule for the task set is shown. In the lower region, allocations within S are shown. These allocations are based on EPDF priorities.*

It can be seen that T misses a deadline at time 10. This is because no quantum is allocated to S in the interval $[5, 10)$. In general, component tasks may be mis-allocated if there exists an interval that contains more component-task windows than supertask windows. Observe that $[5, 10)$ is such an interval; it contains one component-task window and no supertask windows.

Holman and Anderson subsequently showed that such deadline misses can be avoided by inflating supertask weights [22, 23]. While such a scheme is necessarily sub-optimal, experiments presented by them suggest that inflation factors should be small in practice.

In work on real-time systems, the fact that deadlines can be missed has been seen as a shortcoming of supertasking. However, as we have stressed before, deadline misses in routers are not a serious problem, provided reasonable tardiness thresholds can be established. Thus, supertasking may be a very viable approach in this setting. To determine if this is so, we are currently trying to derive tight tardiness thresholds for EPDF-scheduled systems in which supertasks are used.

4.2 Flexibility Issues

In order to reallocate spare capacity in a dynamic Pfair-scheduled system, tasks need to be *reweighted*. In particular, when spare capacity increases (for instance, when a flow becomes inactive or departs the network), the weights of all the active flows should be scaled *up* so that all processing capacity in the system is utilized; when spare capacity reduces due to the arrival of packets in a new or previously inactive flow, the weights of active flows should be scaled *down* in order to accommodate the new flow (though no flow should be scaled below its minimum guaranteed share). When using PD^2 or one of the other optimal algorithms described in Sec. 2.2, weight changes can cause tie-breaking information to change, as shown in Fig. 6. In the worst case, this may require completely resorting the scheduler's priority queue at a $\Theta(N \log N)$ cost, where N is the number of flows. This cost might be incurred *every time a flow becomes active*. Below, we discuss two approaches for avoiding this complexity.

Fast-reweighting approach. Under EPDF scheduling, a *fast reweighting* procedure can be used in which each task's next deadline is preserved. In particular, suppose that task T needs to be reweighted at time t. Let T_i be the subtask of T that is eligible at t. A task T can be reweighted by simply replacing it by a new task U with the new weight and by aligning U_1's window so that $d(U_1) = d(T_i)$ and $e(U_1) \leq t$. (In practice, T's weight can simply be redefined, instead of creating a new task.) This ability to perform

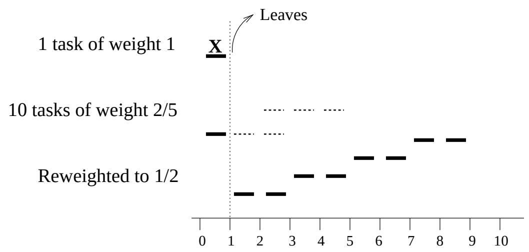

Figure 6: A reweighting example on five processors. At time 0, there are ten tasks of weight $2/5$ and one task of weight 1. At time 1, the latter task leaves. The excess capacity of 1 is redistributed among the remaining ten tasks, giving each a new weight of $1/2$. The dotted lines indicate the original windows of those tasks. The new windows can be aligned so that the new and old deadlines match. The *b*-bit of the first subtask of a task of weight $2/5$ is 1, whereas a task of weight $1/2$ has no subtask with a *b*-bit of 1. Thus, the PD^2 priorities of these subtasks differ.

fast reweighting is another key virtue of EPDF. The fast reweighting approach, however, requires $\Theta(N)$ time, which may be unacceptable in high-speed routers that handle a large number of flows.

Virtual-time approach. The concept of *virtual time* is central to many previously-proposed fair link- and uniprocessor-scheduling schemes. Such schemes typically associate a virtual time function with the shared resource (processor/link): at each "real" time instant t, the virtual-time value $V(t)$ reflects the amount of load upon the resource thus far. For instance, if the current load is half the resource's capacity, then $V(t)$ increases at twice the rate of "real" time: $\frac{d}{dt}V(t) = 2$. Using virtual time, these uniprocessor scheduling schemes are able to achieve exactly the effect of the fast reweighting procedure discussed above by merely varying the rate of change of $V(t)$. Hence, when a connection/task enters or leaves the system, reweighting is mimicked in *constant* time rather than the $\Theta(N)$ time required for explicit reweighting.

For various reasons (some of which are discussed in [5]), the concept of virtual time does not generalize directly to multiprocessor systems. The main problem here is that different task weights may need to be scaled by different factors,[5] due to the fact that no task's weight may exceed unity. A heuristic for dealing with this problem that requires $\Theta(M)$ time is given in [13]. We are currently trying to develop a notion of virtual time that can be used to mimic the effect of reweighting in only constant time.

4.3 Performance Issues

As mentioned earlier, deadline-based scheduling algorithms require routers to maintain a sorted queue of tasks, the complexity of which is a function of the number of packets (or flows). The use of nontrivial tie-breaking mechanisms increases this complexity. The overhead due to tie-breaks is much greater if flows are allowed to utilize idle capacity, as this may introduce the need to re-order sorted task lists (see Sec. 4.2). To enable routers to operate efficiently in high-speed networks, the complexity associated with sorting packets needs to be reduced.

One option is to employ an *approximate* constant-time priority queue implementation that does not sort packets in the exact order of their deadlines, but according to the intervals in which deadlines fall [17, 30]. However, there is no evidence that suggests that meaningful deadline guarantees can be provided with such implementations. Nevertheless, we plan to analyze such implementations as part of our future research.

[5]If quanta of variable length are allowed (as discussed in Sec. 4.1), then additional problems arise. Allowing variable-length quanta is equivalent to allowing subtasks with different execution costs. Because task weights may be scaled by different factors, the order in which such subtasks complete execution in the ideal GPS schedule may change. For example, consider a two-processor system that has three tasks T, U, and V of weights 1/4, 3/4, and 1, respectively. If T and U have subtasks of length 1 and 2 respectively, then the initial finishing times in the GPS schedule will be 4 and 8/3 respectively. However, if V leaves, then the weights of both T and U are increased to 1 in order to fully utilize the two processors. The finishing times of the subtasks of T and U now become 1 and 2 respectively, thus resulting in a change in the order of finishing times. Since tardiness bounds for the actual schedule are determined based on these GPS finishing times, it is not clear whether reasonable bounds can be obtained.

4.4 Scalability Issues

A number of end-to-end link-scheduling frameworks have recently been proposed that enable networks to provide end-to-end per-flow guarantees with respect to shared-link access, without requiring per-flow state in core routers [27, 37, 40]. Over the past year, we have implemented routers from core-stateful and core-stateless networks on Intel's IXP1200-based router platform and compared their performance with that of conventional IP routers [21]. We have found that core routers in stateful networks may be able to process packets at less than 50% of the processing rates of current IP routers, whereas those in core-stateless networks can operate within 75% of these routers. Thus, core-stateless link-scheduling frameworks significantly improve the link speeds at which core routers can operate. This concept has not yet been applied to the problem of processor scheduling in routers.

To see the need for per-flow state in Pfair-based multiprocessor scheduling algorithms, observe that the deadline for processing a packet (or a subtask, in Pfair terminology) depends on its eligibility time, which in turn is a function of the deadline of the *previous* packet of the same flow. Thus, the latest deadline used within each flow must be stored (per-flow state).

As shown by Kaur and Vin [28], *upper bounds* on these latest deadlines can be computed based on the state of the *same* packet at the first node on the end-to-end path of the flow. Since the first node is an edge router of the network, it maintains per-flow state, and can compute deadlines. The edge router can communicate these deadlines to core routers by encoding them in packet headers. Kaur and Vin showed that when core routers use such upper bounds, instead of actual deadlines, guarantees on end-to-end delays remain unchanged. They also showed that core-stateless networks can be designed to provide end-to-end throughput and fairness guarantees as well [27, 29]. We believe that the need for per-flow state in processor scheduling can also be eliminated by using similar state-encoding and deadline-computation techniques. We are cur-

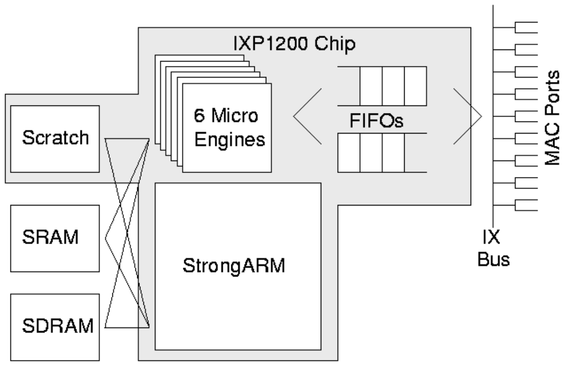

Figure 7: Block diagram of the IXP1200 [34].

rently trying to obtain various end-to-end guarantees that are applicable to packet processing and analyze the guarantees that can be provided in a core-stateless network.

4.5 Other Issues

Recent advances have provided architecture designers with the ability to implement simultaneous multithreading (SMT) in a single superscalar processor [15]. SMT is a technique that allows several independent threads to issue instructions to different functional units in a single cycle. An SMT processor can thus dynamically partition its resources among several simultaneously active threads on a cycle-by-cycle basis. The maximum number of simultaneously active threads is determined by the number of available hardware contexts.

SMT-based systems benefit from schedulers that take into account the various functional units accessed by threads, when trying to schedule threads. Recent results by Jain *et al.* suggest that deadline-based algorithms are capable of making effective use of SMT [24]. However, more research is needed to determine the best approach for allocating resources in combination with our scheduling algorithms, which may need to be adapted to take advantage of SMT.

5 Experimental Evaluation

The techniques presented in Sec. 4 represent our efforts towards designing multiprocessor scheduling schemes that can provide strong assertions about service guarantees. To assess the performance and scalability of these techniques, a real prototype is required. We plan to use Intel's IXP1200 [1] — a platform with six RISC-based *micro-engines*, each with for four hardware contexts (see Fig. 7) — to instantiate our techniques.

Implementation challenges. Over the past year, we have used the IXP1200-based router platform to conduct a preliminary investigation of the implementation of router building blocks — specifically, IP routing, flow classification, and priority queue maintenance — in popular network architectures. While we expect to leverage from this prior router implementation for instantiating new multiprocessor scheduling mechanisms, we need to address new implementation challenges. The most relevant is the lack of explicit software support for thread scheduling within a microengine. In the software reference design provided with the IXP1200 platform, new threads can run on a microengine only when the current thread voluntarily gives up control of that microengine. To address this challenge, we plan to explore two different scheduling approaches: **(i)** using explicit instructions in the code of each thread that force it to give up control after a certain number of instructions, and to select the next thread to run; **(ii)** devoting one of the threads within each microengine to perform all scheduling functions on that microengine. While the former approach will result in less fine-grained control over quantum durations, the latter will result in greater complexity and overheads due to the need for inter-thread communication.

Metrics of interest. We plan to use the router testbed to measure the following: **(i)** the processing speeds supported by the various scheduling mechanisms we develop and those of the past (*e.g.*, EPDF without reweighting, and the heuristic schemes of [13, 14]); **(ii)** the efficacy of these mechanisms in isolating flows from each other and providing service guarantees; **(iii)** utilization gains achieved by allowing flows to use idle processing capacity. These measurements depend on several factors, such as traffic load, network topology, and the processing architectures of routers. We plan to repeat our measurements for different settings for these parameters.

6 Summary

Two important trends are expected to guide the design of next-generation networks. First, with the commercialization of the Internet, providers will use value-added services to differentiate their service offerings from other providers; such services require the use of sophisticated resource scheduling mechanisms in routers. Second, to enable extensibility and the deployment of new services in a rapid and cost-effective manner, routers will be instantiated using programmable network processors. In this paper, we have focused on the problem of scheduling processing capacity on programmable multiprocessor router platforms. Our contributions are twofold. First, we have identified several issues that arise if existing multiprocessor scheduling schemes are used on routers. Existing fair multiprocessor scheduling techniques have considerable promise in this setting, but need to be refined to address performance, scalability, flexibility, and applicability concerns. Second, we have presented new ideas and planned research directions to address these concerns.

References

[1] Intel®ixp1200 network processor family. http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[2] Intel®ixp2400 network processor. http://www.intel.com/design/network/products/npfamily/ixp2400.htm.

[3] Intel®ixp2800 network processor. http://www.intel.com/design/network/products/npfamily/ixp2800.htm.

[4] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proc. of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[5] J. Anderson, S. Baruah, and K. Jeffay. Parallel switching in connection-oriented networks. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 200–209, Dec. 1999.

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

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

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

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

[10] S. Baruah, J. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proc. of the 9th International Parallel Processing Symposium*, pages 280–288, Apr. 1995.

[11] J. Bennett and H. Zhang. WF^2Q: Worst-case fair queueing. In *Proc. of IEEE INFOCOM*, pages 120–128, Mar. 1996.

[12] J. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, Oct. 1997.

[13] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proc. of the 4th Symposium on Operating System Design and Implementation*, pages 45–58, Oct. 2000.

[14] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers. In *Proc. of the 7th IEEE Real-time Technology and Applications Symposium*, pages 3–14, May 2001.

[15] CNP810SP®. Network services processor: Key feature summary. http://www.clearwaternetworks.com/product_summary_snp8101.html.

[16] K. Coffman and A. Odlyzko. The size and growth rate of the internet. Mar. 2001. http://www.firstmonday.dk/issues/issue3_10/coffman/.

[17] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw Hill, 1996.

[18] A. Demers, S. Keshav, and S. Shenkar. Analysis and simulation of a fair queueing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–12, Oct. 1990.

[19] S. Golestani. A self-clocked fair queueing scheme for high speed applications. In *Proc. of IEEE INFOCOM*, 636–646, Apr. 1994.

[20] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proc. of ACM SIGCOMM*, pages 157–168, Aug. 1996.

[21] B. Hardekopf, T. Riche, J. Kaur, J. Mudigonda, M. Dahlin, and H. Vin. Scalability analysis of software-based service-differentiating routers using network processors. Technical Report, Department of Computer Sciences, University of Texas at Austin, May 2001.

[22] P. Holman and J. Anderson. Guaranteeing Pfair supertasks by reweighting. In *Proc. of the 22nd IEEE Real-time Systems Symposium*, pages 203–212, Dec. 2001.

[23] P. Holman and J. Anderson. Using supertasks to improve processor utilization in multiprocessor real-time systems. In *Proc. of the 15th Euromicro Conference on Real-time Systems*, to appear in July 2003.

[24] R. Jain, C. Hughes, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *Proc. of the 23rd IEEE Real-Time Systems Symposium*, pages 134–145, Dec. 2003.

[25] M. Jones and J. Regehr. CPU reservations and time constraints: Experience on Windows NT. In *Proc. of the 3rd USENIX Windows NT Symposium*, page 93–102, July 1999.

[26] P. Kaiser. A (r)evolutionary technology roadmap beyond today's OE industry. *NSF Workshop on The Future Revolution in Optical Communications and Networking*, Dec. 2000.

[27] J. Kaur. *Scalable network architectures for providing per-flow service guarantees.* Ph.D. thesis, Department of Computer Sciences, University of Texas at Austin, Aug 2002.

[28] J. Kaur and H. Vin. Core-stateless guaranteed rate scheduling algorithms. In *Proc. of IEEE INFOCOM*, pages 1484–1492, Apr. 2001.

[29] J. Kaur and H.M. Vin. Core-stateless guaranteed throughput networks. In *Proc. of IEEE INFOCOM*, to appear in Apr. 2003.

[30] J. Liebeherr and D.E. Wrege. Priority queueing schedulers with approximate sorting in output buffered switches. *IEEE Journal on Selected Areas in Communications*, 17:1127–1145, June 1999.

[31] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 294–303, Dec. 1999.

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

[33] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 216–229, Oct. 2001.

[34] T. Spalink, S. Karlin, and L. Peterson. Evaluating network processors in IP forwarding. Technical Report TR-626-00, Department of Computer Science, Princeton University, Nov. 2000.

[35] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Proc. of the 15th Euromicro Conference on Real-Time Systems*, to appear in July 2003.

[36] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proc. of the 34th ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[37] I. Stoica. *Stateless core: A scalable approach for quality of service in the internet.* Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, Dec. 2000.

[38] I. Stoica, H. Zhang, and T. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *Proc. of ACM SIGCOMM*, pages 249–262, Aug. 1997.

[39] L. Zhang. Virtual clock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.

[40] Z. Zhang, Z. Duan, and Y. Hou. Virtual time reference system: A unifying scheduling framework for scalable support of guarantees services. *IEEE Journal on Selected Areas in Communication, Special Issue on Internet QoS*, 18(12): 2684–2695, Dec. 2000.