Master's Thesis

Title

A Study on High-Speed and Scalable Packet Scheduling Algorithm for Achieving Fair Service

Supervisor

Professor Hideo Miyahara

Author

Ichinoshin Maki

February 13th, 2002

Department of Informatics and Mathematical Science

Graduate School of Engineering Science

Osaka University

Master's Thesis

A Study on High–Speed and Scalable Packet Scheduling Algorithm for Achieving Fair Service

Ichinoshin Maki

Abstract

Broadband access technologies such as xDSL and optical fiber have been removing the limits on bandwidth usage of end users and backbone networks could be heavily congested. Thus, it may happen that a limited number of users consume a large amount of network resources and deteriorates quality of other users. Therefore, fair bandwidth usage among end users is becoming more important criteria. The most promising schemes of router mechanisms realizing fair services are per-flow traffic managements, such as per-flow scheduling or per-flow accounting. Although these mechanisms must be needed even in backbone routers to cope with broadband access technologies, it is difficult to be implemented because per-flow state management is prohibitive in such a high-speed environment.

In this thesis, we propose a scalable packet scheduling scheme to improve per-flow fairness in high-speed networks. The proposed scheme allows a scalable queue management according to the router capability; for example, it provides per-flow queue management in relatively slow edge routers and allows flow aggregations as the router capacity increases. In the proposed scheme, per-flow fairness degraded by the flow aggregation is improved by estimating the number of flows aggregated into the same queue and allocating bandwidth to the queue proportionally. To realize our scheme in a high-speed environment, we also propose a flow count estimation scheme, which uses a smaller number of states than the number of active flows instead of requiring any precise per–flow state managements. In addition, our scheme preferentially drops the packets of flows having higher arrival rates to improve fairness in the same queue. We evaluate the proposed scheme through extensive simulation studies and show that our scalable scheme can provide per-flow fair service even in large capacity core routers having very high line speeds and many simultaneously active flows. Finally, we implement the proposed scheme on the Intel IXP1200 network processor and discuss the implementation design issues for high–speed routers. The experimental measurement studies show that the proposed scheme provides fairness equivalent to the conventional packet scheduling scheme in relatively slow edge routers where it is allowed to hold per–flow states and provides excellent fairness in high–speed routers even where an available amount of memories is limited.

Keywords

Fairness

Packet Scheduler

QoS

Scalability

Network Processor

Contents

List of Figures

1 Introduction

Fair service among users is already one of the most important goals of those concerned with the quality of best effort traffic, and it is becoming more important as broadband access technologies such as ADSL (Asymmetric Digital Subscriber Line) and FTTH (Fiber To The Home) remove the limits on a person's use of network resources. When available bandwidth of each end user in access lines is very small and such users' traffic into backbone networks is restricted, it is not so a big problem to provide fair service. However, according to rapid deployment of broadband access technologies, end users have not already been concerned about their bandwidth usage and their traffic made backbone networks heavily congestion. That is, the behavior of end users has a direct influence on backbone networks. Particularly, heavy users may utilize a large amount of network resources and deteriorate quality of other users extremely [1]. As a result, it is inevitable for ISP (Internet Service Provider) to build up backbone network lines and general users may be owed its cost. In addition, there are other many problems; for example, in current mainly used TCP (Transmission Control Protocol), it cannot provide fair service for all flows in a heterogeneous communication environment (RTT (Round Trip Time) or line speed), different TCP versions, a coexistent environment with UDP (User Datagram Protocol) flows and illegal operation by code alteration [2, 3].

There are two main packet scheduling schemes for providing per–flow fair service as the router mechanisms. Random Early Detection (RED) [4] and Stabilized RED (SRED) [5] are represented as the first main packet scheduling scheme. These schemes have a single FIFO (First Input First Out) queue and randomly discard packets without requiring per–flow state. These take an advantage of easy hardware implementation but can not provide per–flow fair service for all users. As the second main packet scheduling scheme, per–flow scheduling or per–flow accounting are represented. In per–flow scheduling, a queue is allocated to each flow and a packet scheduler is responsible for transmitting packets from these queues fairly. There are a lot of packet scheduling algorithms but the Deficit Round Robin (DRR) scheme [6] should be one of the easiest to accomplish the per–flow service. In per–flow accounting, on the other hand, all flows share a single queue but are accounted separately: the behaviors of all flows are monitored and the bandwidth usage of these flows is equalized by preferentially discarding the packets of aggressive flows. Flow Random Early Detection (FRED) [7], for example, adjusts the loss rates of active flows according to their buffer occupancy. Neither the scheduling nor accounting schemes, however, are suitable for high–speed core routers because they must maintain the states of all active flows.

Generally, the line speeds of edge routers are lower than those of core routers, and it should not be difficult to handle a lot of flow states against all flows in edge routers. This difference in line speeds is exploited by Core–Stateless Fair Queueing (CSFQ) [8, 9] or Rainbow Fair Queueing (RFQ) [10], in which edge routers and packet headers maintain flow states but core routers do not. The edge routers measure the arrival rate of each flow and write the rate into an extension of the packet header, and the core routers preferentially discard the packets according to the rates specified there. Since CSFQ requires a header extension, though, it cannot be used unless all routers in the domain are replaced by CSFQ routers.

In this thesis, we propose a new scalable packet–scheduling scheme called Hierarchically Aggregated Fair Queuing (HAFQ). When the line speed of a router is low enough that a large number of flow states against all flows can be maintained, HAFQ uses per–flow queuing. And when the line speed increases and the number of maintainable flow states against all flows decreases, it aggregates flows into a small number of queues. It provides per–flow fair service by estimating the number of flows aggregated in a queue and allocating bandwidth to the queue proportionally. It improves fairness among flows aggregated into the same queue by identifying those having higher arrival rates and preferentially dropping their packets. Note that in this scheme the number of queues in a router does not depend on the number of active flows, classes, or ports but is a design choice. The more queues a router has, the better the fairness it can provide.

Another advantage of our scheme is that it requires no flow identification; whereas per–flow scheduling and accounting schemes require a complex mechanism for flow identification. Queue assignment can be implemented by simple hashing methods because the assignment has only to guarantee that packets of the same flow are stored in the same queue. Flow identification is not required even in edge routers performing near per–flow queuing because our scheme allows two or more flows to occasionally be aggregated in the same queue. We evaluate the proposed scheme through extensive simulation studies.

In general, a scheduling complexity can be evaluated if the scheduling algorithm is given. However, its quantitative complexity of hardware implementation can be fully investigated neither by simulation nor by theoretical studies. Accordingly, we implemented our proposed scheme on the Intel IXP1200 network processor [11]. The network processor is recently paid much attention, which is designed to perform high forwarding operations of packets by the parallel hardware contexts allowing to fully utilize the available memory bandwidth. Also, since a network processor is programmable and it can realize many router mechanisms, we can evaluate the proposed scheme in a nearly actual environment. In the evaluation, we evaluate the proposed scheme in a relative slow network environment. However, we show that the proposed scheme processes packets at high–speed equivalent to the conventional packet scheduling scheme and provides excellent fairness in case of the limited memory storage. Therefore we believe that the proposed scheme can be applied to the high–speed networks.

The remainder of the paper is structured as follows. In the next section, we propose a new scalable packet scheduling scheme. In section III, we evaluate the proposed scheme through extensive simulation studies. In section IV, we discuss the implementation design issues of the proposed scheme on IXP1200 and evaluate its scheduling complexity through experimental measurements. Finally, we conclude in section V.

2 Proposal of a High–Speed and Scalable Packet Scheduling Scheme

In this section , we propose a new packet scheduling scheme for providing per–flow fairness. We call the proposed scheme Hierarchically Aggregated Fair Queuing (HAFQ).

2.1 Hierarchically Aggregated Fair Queuing (HAFQ) Algorithm

The basic mechanism of our new scheme is illustrated in Figure 1. When a packet arrives at the router, a 16-bit CRC hashing function assigns it to a queue. It is because it can perform good load balancing [12] that we use its function. Then the active number of flows in each queue is estimated and bandwidth is allocated proportionally. The number of flows is estimated by using a zombie list [5], which is a short history of newly arrived flows. This means that the number of flows can be estimated without maintaining the states of all active flows. And because the zombie list also helps identifying flows whose having high packet arrival rates, fairness among flows in the same queue can be improved by dropping those packets preferentially.

Next, we show the packet output operations. First, transmitting queue is determined by the DRR scheduling. At this time, the deficit counter is nothing or packets are not inserted to all the transmitting queues, but our scheme allocates bandwidth to each queue proportionally to the estimated number of flows. Finally, our scheme forwards a packet from the transmitting queue.

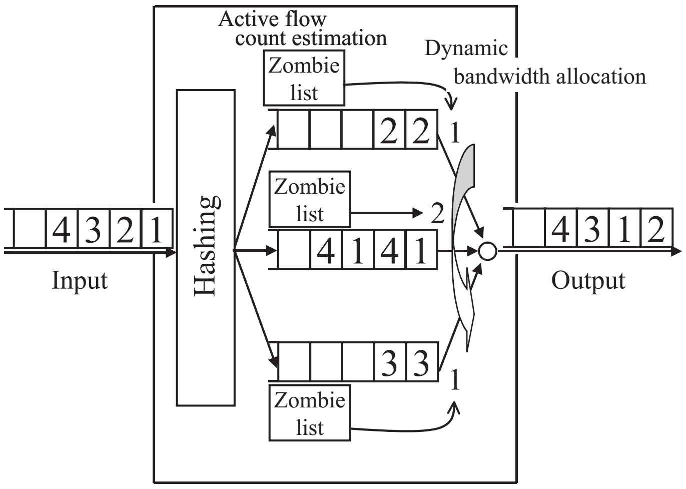


Figure 1: Outline of the proposed scheme.

2.2 Zombie List

A zombie list is a table of constant size in which is a short history of newly arrived flows is kept. Each row in this table contains a *flow ID* and a *packet counter*, and the list is revised every time a packet arrives at the router. An entry in the zombie list is called *zombie*. When a packet arrives at the router, it performs as below.

- Searching into a zombie list.
  - When the flow ID of the packet matches a flow ID in the list, the corresponding packet counter is increased by one. This is called *Hit*.
  - When no entry matches, a row is selected randomly.
    - With probability q, the flow ID of the new packet is written into that row and the corresponding packet counter is set to 1. This is called *Swap*.
    - Otherwise (i.e., with probability $1 - q$), nothing is done. This is called *No-Swap*.

Figure 2 shows an example in which the flow ID of an arriving packet matches the second entry in the zombie list and the corresponding packet counter is incremented by 1 (Fig. 2(a)). It also shows examples of what happens when the new flow ID does not match any of the entries in the zombie list. With probability q, the flow ID is written into a randomly selected entry and the corresponding packet counter is reinitialized (Fig. 2(b)). With probability $1 - q$, the zombie list is not changed (Fig. 2(c)).

SRED [5] uses the zombie list for estimating the number of flows and detecting misbehaving flows. But even though it is possible that one flow is registered in multiple entries, SRED examines only one of the entries in the list. This is an inefficient way to use the list. Our scheme, on the other hand, uses the list efficiently because it searches all entries. And since our scheme maintains multiple zombie lists and each list is small, searching of all entries incurs little computational overhead. In the work reported in this thesis, each zombie list has only four entries.

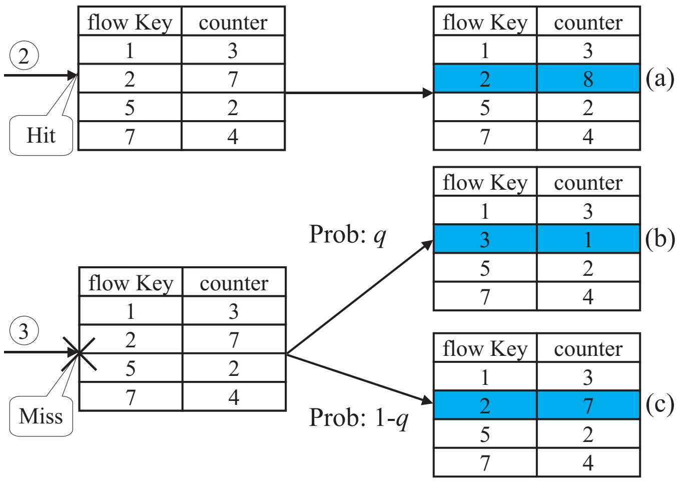


Figure 2: Zombie list.

2.3 Estimating the Number of Flows

In [5], a scheme which estimates the number of flows aggregated in a queue is proposed but the actual number of flows is estimated accurately only when the arrival rates of all flows are equal. Otherwise, the estimated number is too small.

We therefore use a more accurate estimation scheme that works appropriately even when the arrival rates of flows differ, which is common in an actual situation. The arrival rates of incoming flows are estimated and their average is calculated. The number of flows can be derived from the average rate because there is the following relation between the average arrival rate λ_{avg} and the number of flows N.

$$\begin{aligned} \lambda_{avg} &= \frac{\sum_{i=1}^{N} \lambda_i}{N} \\ N &= \frac{\sum_{i=1}^{N} \lambda_i}{\lambda_{avg}} \end{aligned} \tag{1}$$

where λ_i is the arrival rate of flow i. The above equations indicate that the number of flows can be estimated by dividing the total arrival rate by the average arrival rate. Note that these equations hold when the arrival rates of the flows differ.

Now we define R_i as the ratio of the arrival rate of flow i to the total arrival rate for the same queue, i.e.,

$$R_i = \frac{\lambda_i}{\sum_{i=1}^{N} \lambda_i} \tag{2}$$

In the following, we will estimate the number of flows by deriving R_i. Here we assume that the packet length is fixed, but the scheme is easily extended to handle variable packet lengths.

Assume that a packet of flow i arrives at queue k and that zombie list k is updated. Let M denote the number of entries in a zombie list. If entry j ($1 \leq j \leq M$) is replaced by a newly arrived flow, the arrival rate of the flow that has been registered in that entry is estimated by using the packet counter value of the entry before the entry is replaced. This is because the maximum value of the packet counter is proportional to the rate of the flow. When we define P_1 as the probability that a flow in one entry is replaced before packets of the flow arrive (i.e., the probability that the maximum value of the packet counter is 1), P_1 is given by

$$\begin{aligned} P_1 &= (1-R_i)a + (1-R_i)(1-a)(1-R_i)a \\ &\quad + \{(1-R_i)(1-a)\}^2(1-R_i)a \\ &\quad + \cdots + \{(1-R_i)(1-a)\}^n(1-R_i)a \\ &= \frac{(1-\sum_{j=1}^{M} R_{X_j})\frac{q}{M}}{(1-\sum_{j=1}^{M} R_{X_j})\frac{q}{M} + R_i} \end{aligned} \tag{3}$$

where X_j denotes the flow ID registered in entry j and a denotes the probability that an entry is replaced by a newly arrived flow under the condition that an arrived packet matches no entry. Therefore,

$$a = \frac{1-\sum_{j=1}^{M} R_{X_j}}{1-R_i} \times \frac{q}{M}. \tag{4}$$

In the same way, the probability P_n that the packet counter is increased to n before the entry is replaced. P_n is given by

$$\begin{aligned} P_n &= R_i^{n-1}P_1 + (1-R_i)(1-a)R_i^{n-1}P_1 \\ &\quad + \{(1-R_i)(1-a)\}^2 R_i^{n-1}P_1 \\ &\quad + \cdots + \{(1-R_i)(1-a)\}^n R_i^{n-1}P_1 \\ &= \frac{R_i^{n-1}(1-\sum_{j=1}^{M} R_{X_j})\frac{q}{M}}{\{(1-\sum_{j=1}^{M} R_{X_j})\frac{q}{M} + R_i\}^n} \end{aligned} \tag{5}$$

Therefore, the expectation E_i for the maximum value of the packet counter is given by

$$E_i = \sum_{i=1}^{\infty} i\,P_i = \frac{(1-\sum_{j=1}^{M} R_{X_j})\frac{q}{M} + R_i}{(1-\sum_{j=1}^{M} R_{X_j})\frac{q}{M}} \tag{6}$$

Now let R_i be unknown and let $\widetilde{R}_i$ denote the estimation for R_i. If we assume the packet counter value reaches $\widetilde{E}_i$ before the entry is replaced, $\widetilde{R}_i$ can be derived using Eq. (6) as follows:

$$\widetilde{R}_i = \left(1 - \sum_{j=1}^{M} R_{X_j}\right) \frac{q}{M}(\widetilde{E}_i - 1) \tag{7}$$

If we assume that no entries in the zombie list are swapped, the probability that incoming packets match one of the entries (i.e., the probability of a Hit) approaches the sum of the rates of the flows in the zombie list: $\sum_{j=1}^{M} R_{X_j}$. Therefore, if we choose the smaller value for the swapping probability q, the sum of R_{X_j} can be approximated by the probability p. Thus, $\widetilde{R}_i$ can be derived by the following equation:

$$\widetilde{R}_i = (1 - p)\frac{q}{M}(\widetilde{E}_i - 1) \tag{8}$$

Then the scheme computes the average of $\widetilde{R}_i$. Since a flow having a higher arrival rate is counted to the average arriving rate more frequently than other flows, the average is overestimated if some of the flows have higher arrival rates. Since flow i is registered in the zombie list $\widetilde{R}_i/\widetilde{E}_i$ times per unit time, $\widetilde{R}_i$ should be counted into the average with the weight $(\widetilde{R}_i/\widetilde{E}_i)^{-1}$. Therefore, the average $\widetilde{R}_{avg}$ is given by

$$\widetilde{R}_{avg} = \left\{1 - \beta\left(\frac{\widetilde{E}_i}{\widetilde{R}_i}\right)\right\} R_{avg} + \beta\left(\frac{\widetilde{E}_i}{\widetilde{R}_i}\right)\widetilde{R}_i \tag{9}$$

where β is a smoothing parameter for the average. Finally, the estimated number of flows accommodated in the queue is calculated by $1/\widetilde{R}_{avg}$ using Eqs. (1) and (2).

$$N = \frac{1}{\widetilde{R}_{avg}} \tag{10}$$

If the number of flows is no more than the number of entries in a zombie list because all incoming packets are matched with one of the entries, the packet counter can increase infinitely and errors in the estimation increase. Therefore, we introduced another mechanism to deal with this problem.

2.4 Packet Dropping Using Packet Counters

Our scheme improves fairness among flows aggregated in the same queue by detecting the flows having higher arrival rates and preferentially dropping the packets of these flows. Since Eq. (8) shows that the packet counter value is roughly proportional to the packet arrival rate, the packets of flows having higher arrival rates should be dropped preferentially. The proposed scheme therefore drops the incoming packet if the packet counter value is more than the average of the packet counter value and the queue length is greater than half of the buffer capacity.

$$E_i > \frac{\sum_{i=1}^{M} E_i}{M} \tag{11}$$

3 Simulation Results

3.1 Case of Single Bottleneck Link

In the work reported in this subsection we used the single–bottleneck network topology shown in Fig. 3. The bandwidth of the access links and the bottleneck link is 155 Mbps, and the propagation delays of these links are respectively 0.1 and 1 ms. All hosts use TCP (Tahoe) or UDP (3.2 Mbps) and they have an infinite amount of data to transmit. The number of entries in one zombie list is four. All simulations were run using the NS simulator [13].

3.1.1 Estimated Number of Flows

We evaluated the flow number estimation of our new scheme and compared it with the estimation of SRED. We also evaluated the effectiveness of packet–dropping in our scheme. Figures 4(a)–(c) show the estimated number of flows aggregated in a queue and the number of active flows. In these figures, “HAFQ w/o DROP” denotes our scheme without packet–dropping using packet counters and “HAFQ” denotes our scheme with packet–dropping. We assumed that one flow starts to transmit at time 0, that the number of flows doubles every 2 seconds until it reaches 64 and that all these flows are aggregated in one queue.

In Fig. 4(a), all flows are TCP flows and their RTTs are same. In this case, all three schemes

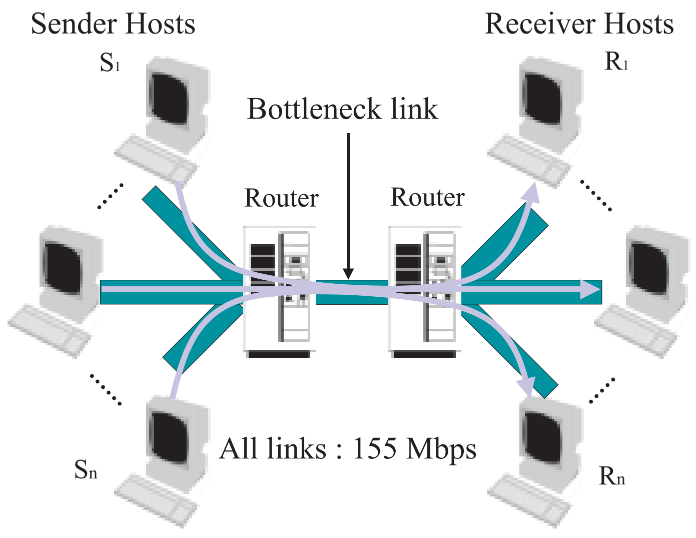
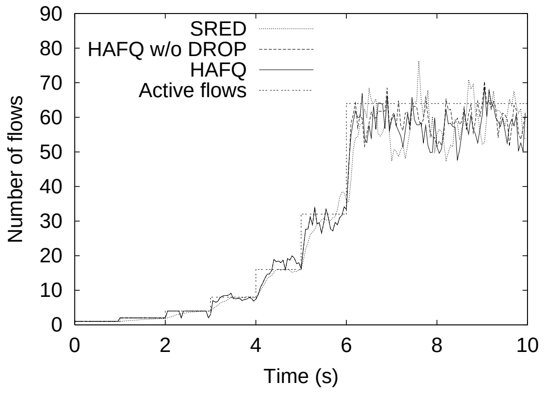


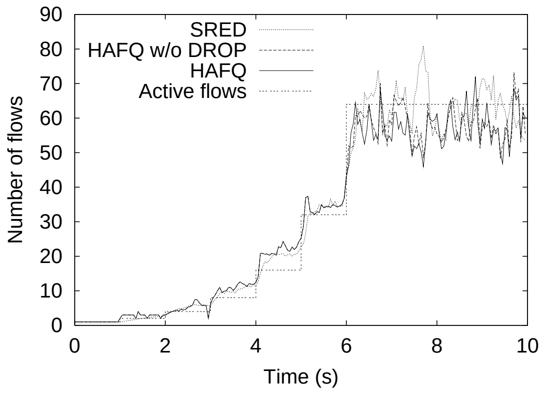
Figure 3: Single-link model.

give approximately correct numbers. In Fig. 4(b), half of the access links have 1 ms propagation delays and the other half have 0.1 ms propagation delays. This figure shows that RTTs have little influence on the estimated number of flows.

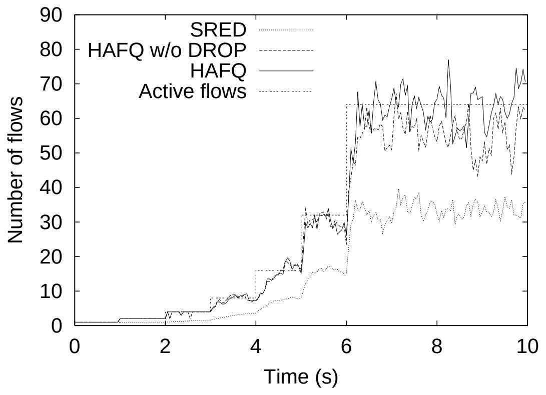
In Fig. 4(c), half the flows are TCP flows and the other half are UDP flows. In this case, the numbers of flows estimated by SRED are much smaller than the correct values. Since the arrival rates of UDP flows are much higher than those of TCP flows, SRED counts flows as if the only flows in the network are UDP flows. In our scheme without the packet–dropping, however, the estimated numbers of flows are only slightly less than the correct values. With the packet–dropping, the estimation is improved and the error is less than 10%. This is because packets of UDP flows are preferentially dropped and the arrival rates of all flows become more uniform.



(a) TCP flows in a homogeneous environment



(b) TCP flows in a heterogeneous environment



(c) TCP and UDP flows in a homogeneous environment

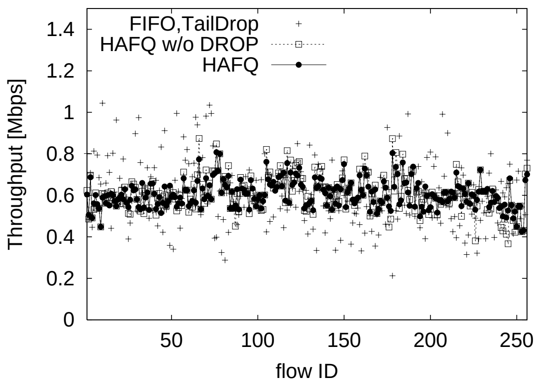
Figure 4: The estimated number of flows in SRED and the proposed scheme.

3.1.2 Throughput of Each Flow on the Middle Aggregation Level

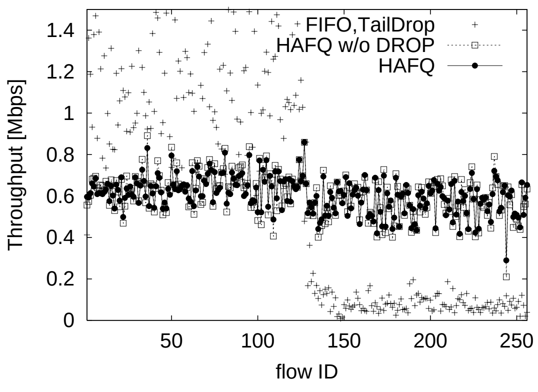
We next evaluated our new scheme using 64 queues and determining the ones that flows were stored in by hashing the flow IDs, in comparison with the FIFO scheme using tail drop for buffer control. In this evaluation the number of flows was 256. We run each simulation during 10 sec.

Figure 5(a) shows the individual throughput of all TCP flows. This figure shows that the throughput of the FIFO scheme differs and the proposed scheme improves fairness among flows. In Fig. 5(b), half of the TCP flows have longer RTTs as evaluated in Fig. 4(b). The FIFO scheme gives TCP flows with shorter RTTs more bandwidth than those with longer RTTs, and our scheme decreases the difference between flows with different RTTs.

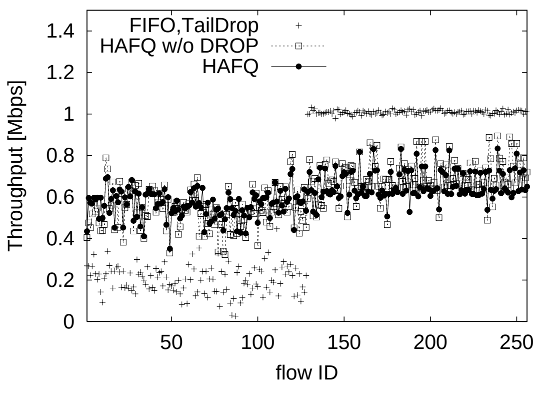
In Fig. 5(c), half of the flows are TCP flows and the other half are UDP flows (1.2 Mbps). In this case, the FIFO scheme gives UDP flows much more bandwidth than TCP flows and our scheme improves fairness between TCP flows and UDP flows. In these three cases, the packet–dropping of our new scheme further improves fairness among flows.



(a) TCP flows in a homogeneous environment



(b) TCP flows in a heterogeneous environment



(c) TCP and UDP flows in a homogeneous environment

Figure 5: Throughput of TCP and UDP flows.

3.1.3 Fairness Index on the High Aggregation Level

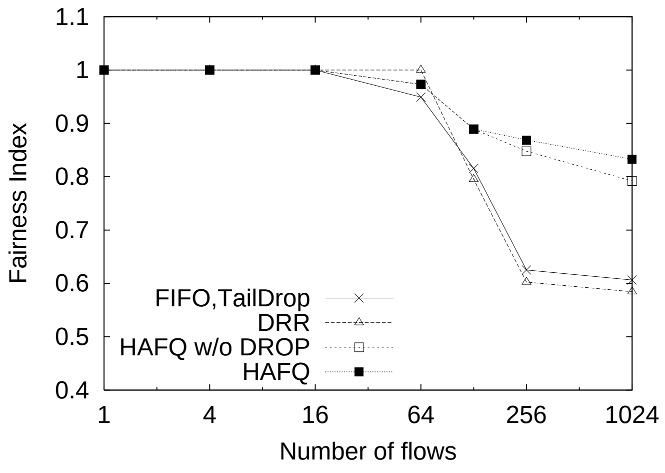
The throughput of a large number of flows was next examined in order to evaluate a fairness property of our scheme in a large–scale–network environment. In this evaluation, our scheme was compared with the FIFO scheme and with a DRR per–flow scheduling scheme. Both our new scheme and the DRR scheme had 64 queues. In the DRR scheme, each flow is accommodated in its own queue if the number of active flows is less than 64; otherwise the extra flows are accommodated in one of the queues randomly for comparison purpose. This is because these extra flows cannot be identified without per–flow information. Here we use a Fairness Index as the fairness measure. Its value f is near 1 if the throughputs of all flows are equal, and it gets smaller as differences in throughput increase. It is calculated as follows

$$f(x_1, x_2, x_3, \cdots, x_N) = \frac{(\sum_{i=1}^{N} x_i)^2}{N \sum_{i=1}^{N} {x_i}^2} \tag{12}$$

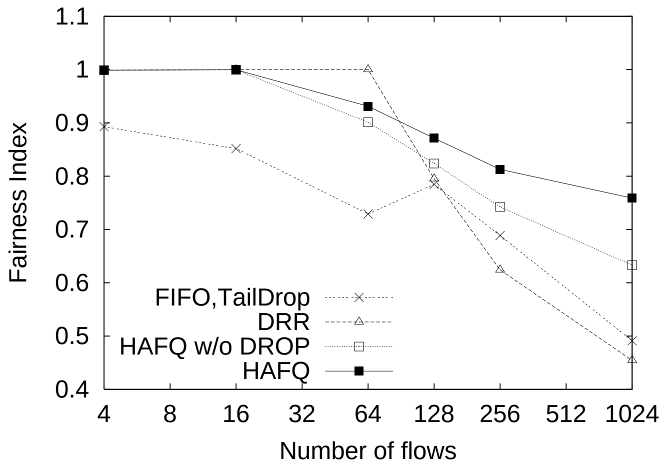
where x_i is the throughput of flow i, and N be the total number of flows.

Figures 6(a) and 6(b) show the fairness index plotted against the number of flows. In Fig. 6(a) all flows are TCP flows, and in Fig. 6(b) half of the flows are UDP flows. The fairness of FIFO scheme becomes worst as the number of flow increases and, compared to the other schemes, its fairness is worst in any situations. The DRR scheme provides excellent fairness if the number of flows is less than the number of queues (i.e., 64), but it becomes increasingly less fair as the number of flows increases beyond 64. This is because there is a difference in the number of flows aggregated in one queue, while the same amount of the bandwidth is allocated to each queue. On the other hand, since our new scheme dynamically allocates bandwidth in proportion to the estimated number of flows, its fairness index decreases only gradually as the number of flows increases.

When the number of flows is 64, the fairness index of our new scheme is worse than that of the DRR scheme. This is because exactly one flow is accommodated in one queue in the DRR scheme, whereas flows are accommodated in the queues randomly in our new scheme. For most numbers of flows, however, our scheme provides better fairness and its fairness is less sensitive to the number of flows. These figures show that the packet–dropping is especially effective when there are many ill–behaved flows in the network.



(a) Only TCP flows



(b) TCP and UDP flows

Figure 6: Fairness index versus the number of flows.

3.2 Case of Multiple Congested Links

In the work reported in this subsection, we used the hierarchical network topology shown in Fig. 7. The bandwidth of the access links, the local links, and the backbone link are respectively 1.5, 16–48, and 300–1500 Mbps. And the propagation delays of these links are respectively 0.1, 1, and 1 ms. The number of edge routers is 32 and each edge router accommodates 32 flows, so 1024 flows are aggregated in the bottleneck link. There are 192 UDP flows sent at 1.5 Mbps, and those UDP flows are accommodated every 6 UDP flows in the same edge routers. The other 832 flows are TCP (Reno) flows. Both edge routers and core routers have enough queues for DRR per–flow queuing. In the proposed scheme, the number of queues which all these routers have is only 64.

The average throughput values of TCP and UDP flows are shown in Figs. 8(a)-(c). The bandwidth of the local links is 16 Mbps in Fig. 8(a), 32 Mbps in Fig. 8(b), and 48 Mbps in Fig. 8(c), respectively. It can be seen that UDP flows always have more bandwidth than TCP flows and are less sensitive to the capacity of the bottleneck link. This means that if the backbone line is not congested and the local links are the bottleneck, the bandwidth of the backbone link has no influence on the throughput (right sides of Fig. 8(a) and Fig. 8(b)) and TCP and UDP flows are served fairly by both the DRR scheme and our new scheme. But if the backbone link is the bottleneck, that is, if the throughput is proportional to the bandwidth of the backbone link, our scheme gives slightly more bandwidth to UDP flows than TCP flows. This is because our scheme aggregates

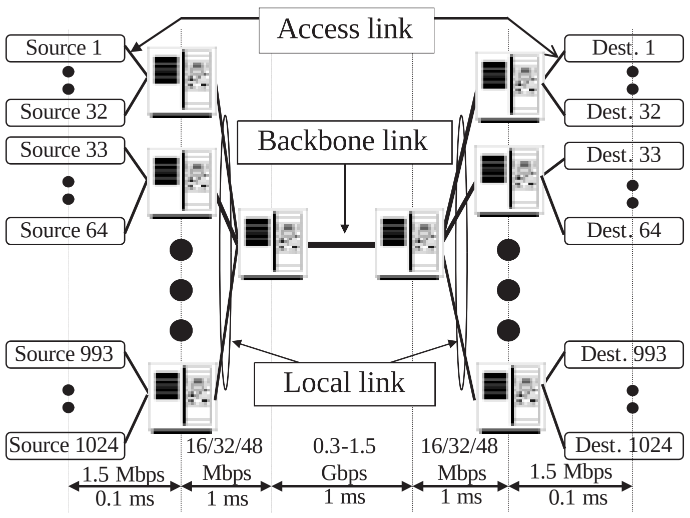
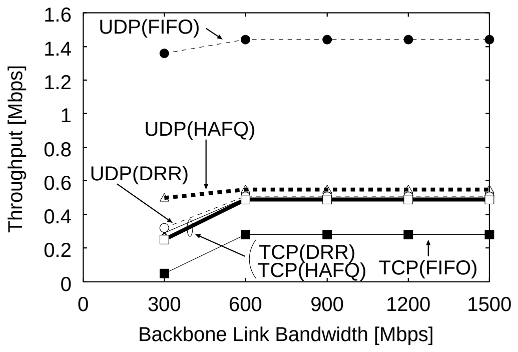


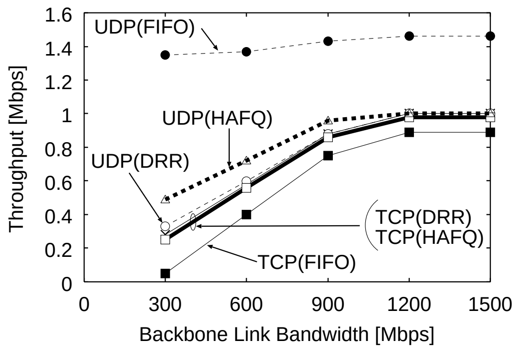
Figure 7: Multiple-links model.

multiple flows in the routers, while the DRR scheme maintains per–flow queues in the routers. Although per–flow DRR scheduling provides excellent fairness among flows, it would be difficult to have per–flow queues in core routers.

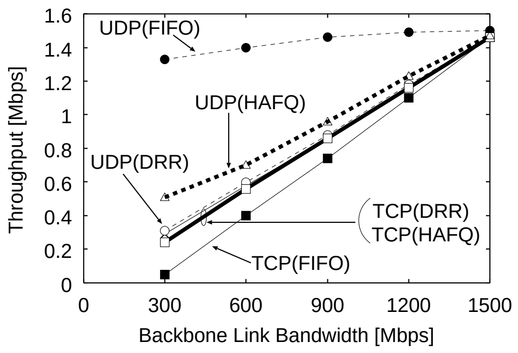
We therefore also evaluated a combination scheme where the edge routers maintain per–flow queues but the core routers each have only one FIFO queue. The throughput of TCP and UDP flows is shown in Fig. 9 for both this combination scheme and our new scheme. In this figure, the label “uniform” means that UDP flows are equally accommodated in all edge routers and “hot spot” means that UDP flows are accommodated only in particular edge routers. In the hot–spot case, six routers accommodate 32 UDP flows and the other edge routers accommodate TCP flows only.Figure 9 shows that both schemes provide fairness among flows if the backbone line is not congested. Otherwise, there is an unfairness between TCP flows and UDP flows even if edge routers have per–flow queues. Our new scheme is fairer than the combination scheme because of the aggregated queuing with flow counts estimation in the core router.



(a) Local links: 16 Mbps



(b) Local links: 32 Mbps



(c) Local links: 48 Mbps

Figure 8: Throughput in each Local link bandwidth.

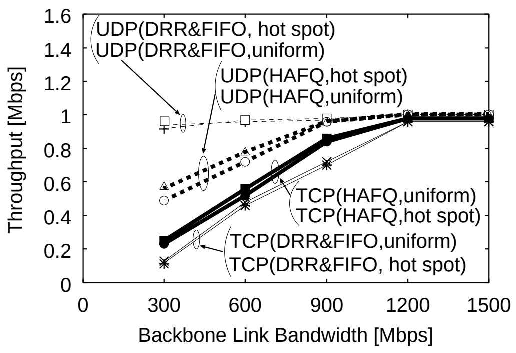


Figure 9: Local links: 32 Mbps.

4 Implementation Design Issues of HAFQ on the IXP1200 Network Processor

In this section, we discuss the implementation design issues of HAFQ on the IXP1200 network processor. Figure 10 illustrates our scheme implemented on IXP1200 [11]. IXP1200 has six microengines for packet forwarding operations, each of which can deal with four threads (contexts) concurrently. A microengine is a 32–bit RISC programmable data engine and a thread can realize multiple control streams in one program. In addition, IXP1200 provides (1) an SDRAM unit to access low cost, high bandwidth memory for mass data, (2) an SRAM unit for very high bandwidth memory to store lookup tables and other data for packet processing, and (3) a scratch pad memory which is an embedded memory unit.

A unique feature of the IXP1200 network processor is that as an incoming IP packet is received, the microengine breaks the packet into 64–byte MPs (MAC–Packets), and often that, the microengine context deals with a MP, not a packet.

4.1 Implementation Outline of the Proposed Scheme on IXP1200 Network Processor

The microengines #1 through #4 perform packet input operations including packet header verification, destination address lookup, header modification and HAFQ ingress operations of IPv4 packets. Then, the microengines #5 and #6 perform packet output operations including determination of the transmitting queue by the DRR scheduling and dynamic bandwidth allocation. This "2–to–1 allocation" is based on the suggestion described in [14]. We use the SDRAM unit to deploy packet data as a shared buffer pool. Since it has larger memory capacity and higher memory bandwidth than the SRAM unit, it is suitable to the shared packet buffer pool. On the other hand, the SRAM unit holds the routing table, zombie list, and other states which are needed in performing HAFQ operations.

4.1.1 Packet Input Operations

Figure 11 illustrates the pseudo–code of packet input operations. When a microengine context receives a packet, it performs packet header operations and HAFQ ingress operations. If the MP is the top 64–byte data of the packet (*Start_Of_Packet*), a flow ID is determined by assigning the hash

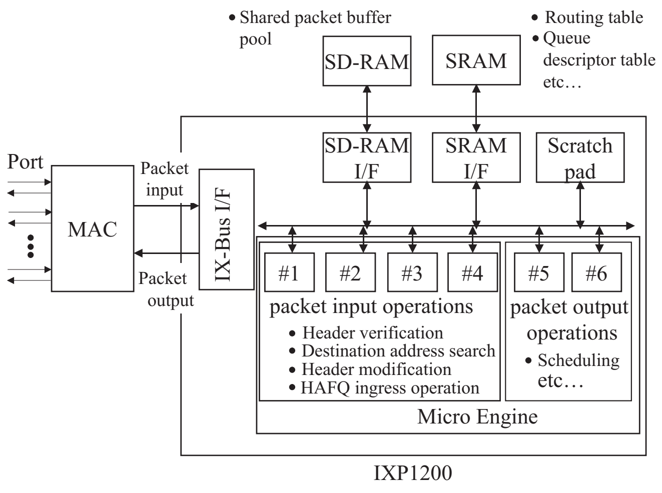


Figure 10: Our task assignment on the IXP1200 network processor.

function to the parameters (source and destination addresses, port numbers of its packet, and so on). See the third line of Fig.11. Then, the packet is allocated associated queue (*Queue*) by using the calculated flow ID as shown on the fourth line of Fig.11. Here, we note that the calculated flow ID is unique to all flows.

Next, the flow Key of the arriving packet is determined as shown on the fifth line of Fig.11 and the microengine context searches a flow ID into the zombie list of the queue determined by the flow ID (the sixth line of Fig.11). A flow Key is a unique identifier in this queue. If the flow Key of the arriving packet matches a flow Key in an entry (*m*), the corresponding packet counter is increased by one and the miss probability is updated (the eighth and ninth lines of Fig.11). If its packet counter becomes larger than the average counter value *(($\sum_{i=1}^{M}$ Counter [Queue, i]) / M)*, drop preference (*drop*) is set (the 11th line of Fig.11). Otherwise, the miss probability is updated in a similar way as described above. Then, with probability q, the microengine context executes the following operations. First, an entry of the zombie list in this queue is selected randomly and the arrival rate of the flow which has been held in this entry is calculated (the 15th line of Fig.11). Next, the average arrival rate of all flows in this queue is updated and the number of active flows that are aggregated in this queue is estimated (the 16th, 17th, and 18th lines of Fig.11). Also, the flow Key of the new packet is written into that entry, and the corresponding packet counter is reinitialized (the 19th and 20th lines of Fig.11).

If the length of the queue is greater than the buffer capacity, or if the drop preference is set and the queue length is greater than the half of the buffer capacity as described in Subsection 2.4, the MP is discarded (the 23th line of Fig.11). Otherwise, the microengine context stores the MP packet data into the shared buffer pool on the SDRAM unit (the 25th line of Fig.11). Finally, receiving *End_Of_Packet* indicates the context that the whole packet has now been received, and the microengine context inserts the pointer of the SDRAM address into the transmitting queue list on the SRAM unit. That is, management of the transmitting queue is performed on the SRAM unit and the SDRAM address is stored there. This is because a stack scheme is realized on the SRAM unit and it is easy to manage the transmitting queue.

```
Input:
 If (Start_Of_Packet)
  flow ID = Hash (SrcAddr, DstAddr)
  Queue = flow ID mod Num_Of_Queues
  flow Key = flow ID / Num_Of_Queues
  m = Search_ZombieList (Queue, flow Key)
  If Hit
   Counter [Queue, m] ++
   $miss$ [Queue] = $miss$ [Queue] $\cdot$ (1-$\alpha$)
   if (Counter [Queue, m] $>$ (($\sum_{i=1}^{M} Counter[Queue, i]$) / M))
    drop = high
  else
   $miss$ [Queue] = $miss$ [Queue] $\cdot$ (1-$\alpha$) + $\alpha$
   If (Random_Number $< q$)
    $R_i$ = miss [Queue] $\cdot \frac{q}{M} \cdot$ (Counter [Queue, random] - 1)
    Avg_ArrivalRate [Queue] = $\{$1 - $\beta \cdot$ ($\frac{Counter[Queue,random]}{R_i}$)$\}$
              $\cdot$ Avg_ArrivalRate [Queue] + $\beta \cdot$ ($\frac{Counter[Queue,random]}{R_i}$) $\cdot R_i$
    Num_Of_Flows [Queue] = 1 / Avg_ArrivalRate [Queue]
    Counter [Queue, random] = 1
    Key [Queue, random] = flow Key
 if ((length [Queue] $>=$ MAX) $||$
        ((drop = = high) && (length [Queue] $>=$ MAX / 2)))
  discard MP
 else
  Store MP to SDRAM
  if (End_Of_Packet) Enqueue (Queue)
```

```
Variables
$m$ :  An entry in a zombie list
$miss$ :  miss probability
$\alpha$, $\beta$ :  smoothing parameter
$M$ :  the number of zombies
Random_Number :  non-negative decimal number less than 1
$q$ :  swap probability
$R_i$ :  the arrival rate of flow $i$ to the rate
        of all flows accommodated to this queue (Queue)
```

Figure 11: Pseudo–code in packet input operations

4.1.2 Packet Output Operations

Figure 12 shows the pseudo–code in packet output operations. First, a next transmitting queue is selected by the DRR scheduling. If the deficit counter is zero or packets are not inserted to all the transmitting queues, the microengine context allocates the bandwidth to each queue proportionally to the estimated number of flows (the 15th through 19th lines of Fig.12). Next, a microengine context reads the top packet data of the transmitting queue from the SDRAM unit. Then the microengine context moves it to the T–FIFO element. A T–FIFO element is the transmitting FIFO buffer and IXP1200 has 16 T–FIFO elements. Then, the deficit counter is updated (the ninth of Fig.12). The microengine context is not concerned with packet data stored in T–FIFO elements and those data is sent to the network.

```
Output:
 j = 1
 if (j $\leq$ Q)
    if ((Deficit_Counter [Queue] $\leq$ 0) ||
           (Queue_Length [Queue] = = 0))
       Queue ++; j ++
    else
       Read MP From SDRAM to T-FIFO
       Deficit_Counter [Queue] = Deficit_Counter [Queue] - MP_size
       Transfer MP
       Queue ++
 else
    j = 1
    while (j $\leq$ Q)
          weight [j] = Num_Of_Flows [j] $\cdot$ quantum
          if (Deficit_Counter [j] $<$ 0)
             Deficit_Counter [j] = Deficit_Counter [j] + weight [j]
          else
             Deficit_Counter [j] = weight [j]
```

Variables

j :	control variable
Q :	the number of queues
Queue :	the queue where a MP is transmitted in the previous turn
MP_size :	data size of the MP which is transmitted
quantum :	fixed size

Figure 12: Pseudo–code in packet output operations

4.2 Estimation on Memory Capacity

Before presenting the evaluation result of our HAFQ implemented on IXP1200, we discuss the required memory capacity, in this subsection. Figure 13 for the memory map. Note that the arguments made in this section is not limited to IXP1200, but to a wider range of router architectures.

4.2.1 Memory Capacity in Zombie List

A zombie list is a table whose row contains a flow Key and a packet counter as described in Subsection 2.2. Since it is a key component of our scheduling algorithm, we start with discussing the implementation design issues of the zombie list.

According to [15], the number of active flows in backbone networks reaches about several tens of thousands. To discriminate those active flows, 16–bit is necessary for identifying the flow. However, our method has a capability of aggregating flows. The required key length is determined by the number of active flows within each queue. We therefore chosen 12–bit for the length of the flow key.

Next, we describe the required memory capacity for the packet counter. Eq.(6) can be rewritten as $E_i < 1 + M/q$. Since we know that $\{q, M\}$ = $\{0.1, 4\}$ or $\{0.01, 4\}$ is an appropriate parameter set from simulation experiments, E_i is bounded by 41 or 401, respectively. Therefore, we allocate 10–bit for a packet counter per an entry in the zombie list. In addition, we introduced a valiable *pointer* to prevent the packet counter value from increasing infinitely. This is because our scheme changes the number of entries in the zombie list dynamically, and there are less entries than the number of flows. Since memory word bandwidth of the SRAM unit is 32–bit, those three variables (flow Key, packet counter, and pointer) can be stored in the same word. Therefore, required memory capacity for the zombie list is determined by $M \times Q \times 32$ bit, where M and Q represent the number of entries in the zombie list and the number of queues, respectively.

4.2.2 Memory Capacity Required for Estimating Number of Flows

To estimate the number of active flows in each queue, our scheme needs the average probability that is stored in the zombie list, and the average arrival rate of flows aggregated to the queue. Since our scheme uses Eq.(8) in estimating the number of active flows, we introduce the average probalbility not updated in the zombie list (*Miss Probability*) in order to reduce required opera-

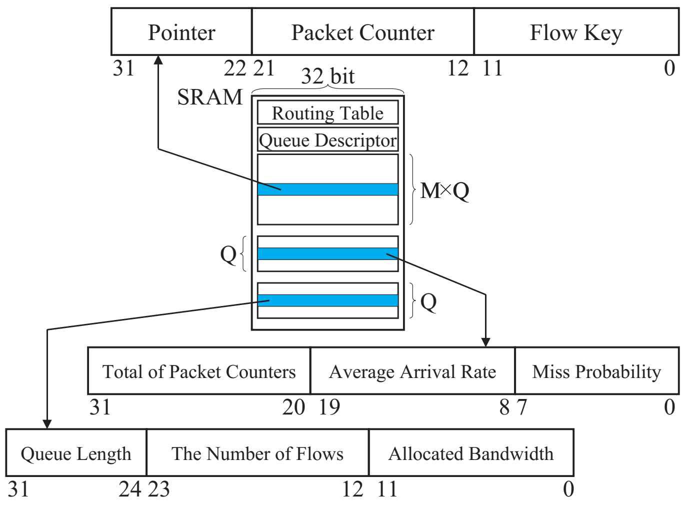


Figure 13: Memory map on the SRAM unit.

tions. In addition, we try to reduce the number of instructions by setting the parameters, such as q or M, to the power of 2. We reserve chosen 8–bit for storing the miss probability that expresses a non–negative value ranging from 2^{-128} to 1.

For the average arrival rate (*Average Arrival Rate*), we use 12–bit. This is because we have the relation $\dfrac{1}{2^{12}} \leq R_{avg} \leq 1$ from the relations $R_{avg} = \frac{1}{\widetilde{N}}$ and $\widetilde{N} \leq 2^{12}$. These variables are also stored in the same word on the SRAM unit. In addition, we store the total of the packet counters (*Total of Packet Counters*) in the zombie list on the same address in order to improve the processing speed. These variables can reduce the required number of instructions.

From the above estimation, the required memory capacity for estimating the number of flows is $Q \times 32$ bit.

4.2.3 Memory Capacity Required for Packet Output Operations

Our scheme determines the transmitting queue by the DRR scheduling. Thus, allocated bandwidth to each queue and the number of flows estimated for the bandwidth allocation algorithm are needed to be maintained in the SRAM unit. From the relation of $\widetilde{N} \leq 2^{12}$, we chosen 12–bit for each of these variables. The queue length is also put in the unused portion of the same word. Therefore, the required memory capacity is $Q \times 32$ bit.

4.2.4 Summary

As a summary, the total required memory capacity for implementing our scheme is given by

$$M \times Q \times 32 + Q \times 32 + Q \times 32 = 32 \cdot Q \cdot (2 + M) \quad [bit] \tag{13}$$

The memory capacity that edge routers can have in the case of using an off–chip memory is up to 4 Mbytes in the current memory technologies. Since our scheme can have about 512K queues, it would be possible to provide per–flow scheduling. On the other hand, core routers use an on–chip memory because the line speed is very high in backbone networks. Since its memory capacity is about 32 Kbytes, our scheme can have 1K queues from Eq.(13).

4.3 Implementation Evaluation

We use the IXP1200 Developer Workbench [16] for simulation. We suppose the network model shown in Figure 14. 96 sender hosts and 96 receiver hosts are connected through the router, and each sender host generates IP packets whose length is a fixed size of 512 byte. Each sender host generates packet at 100 Mbps (CBR), and those sender hosts have an infinite amount of data to transmit. Of course, 9.6 Gbps of input rate in total is too large for the IXP1200 packet forwarding rate. Our intention here is to investigate the maximum packet processing rate of the scheduling algorithm. We last note that our current experimental implementation does not perform packet header modification or routing table lookup, because we wanted to evaluate the overhead of the proposed scheme.

4.3.1 Evaluation on Packet Processing Capacity

In this subsection, we investigate the packet processing capacity of the proposed scheme by comparing to that of the DRR scheme. These two schemes have 16 queues, and additionally HAFQ has two entries in each zombie list.

Figure 15 shows a packet processing capability of packet input operations and output operations. In packet input operations, the processing capability of HAFQ is about 5 percent lower than that of DRR. This is because HAFQ operations require additional instructions such as searching the zombie list and estimating the number of flows. In addition, as the number of flows aggregated increases, the packet receiving capability of HAFQ becomes low. This is because the overheads

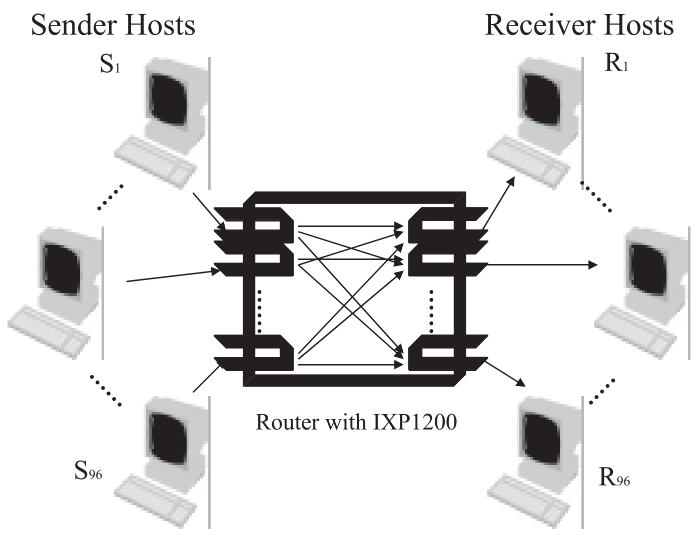


Figure 14: Evaluation model.

for multiplication instructions used in estimating the number of flows are often incurred. As for packet output operations, although the processing capability of HAFQ is lower than that of DRR, the performance degradation is limited.

We notice that the performance degradation of HAFQ operations is mainly caused by multiplication instructions as described above. Such an instruction requires many cycles, and those are often used in our code. IXP1200 has a RISC processor, StrongARM. Since the flow count estimation and bandwidth allocation are not required frequently, the performance of HAFQ operations seems to be improved by performing multiplication instructions at the StrongARM, but its evaluation is left to be a future research topic.

4.3.2 Fairness in the Case of the Different Number of Flows

In this subsection, we evaluate a fairness property of our scheme and the DRR scheme as the number of flows increases. In this evaluation, we use the Fairness Index as the fairness measure (See Eq.(12)). Both our scheme and the DRR scheme have 16 queues, and the number of entries in each zombie list is two in our scheme. In the DRR scheme, each flow is accommodated in its own queue if the number of active flows is less than 16; otherwise, the extra flows are accommodated in one of the queues randomly. On the other hand, our scheme determines the accommodated queue randomly, even if the number of flows is less than 16.

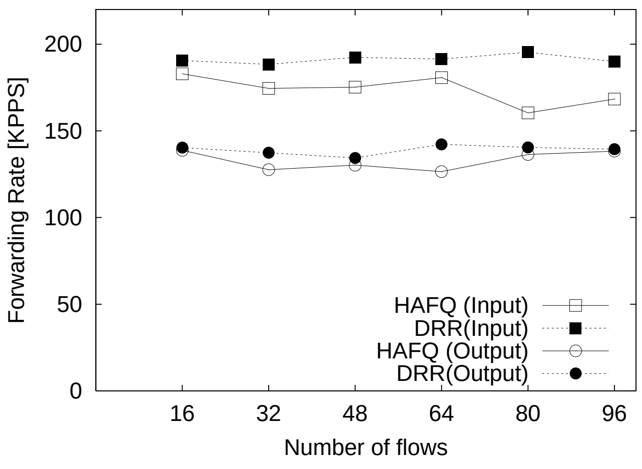


Figure 15: Packet forwarding rate.

Figure 16 shows the fairness index against the number of active flows. The same tendency can be observed as in the simulation experiments presented in Subsection 3.1.3. We can confirm that the DRR scheme provides excellent fairness if the number of flows is equal to the number of queues, but it provides degraded fairness as the number of flows increases. In our scheme on the other hand, although the fairness index decreases gradually as the number of flows increases, its high fairness can be provided.

4.3.3 Fairness in the Case of the Different Memory Capacity

In this subsection, we evaluate the effect of the available memory capacity on the fairness property. In our scheme, the required memory usage greatly depends on both the number of entries and the number of queues. In this subsection, we investigate the changes of the fairness affected by those parameters. Figure 17 shows the fairness index against the available memory capacity. In the DRR scheme, the number of queues is changed according to the available memory capacity. Recalling that 4 byte is necessary for managing one queue, its fairness is improved as the memory size becomes large. However, the fairness of our scheme is larger than that of the DRR scheme as shown in Fig.17. In obtaining the results, the number of entries is fixed at two in each zombie list and the number of queues is changed as 8, 16, and 24. In Fig.17, it is labelled by "HAFQ / queue". That is, required memory capacity is 128, 256, and 384 bytes. The fairness index is improved as the available memory capacity is large. This is because the number of flows aggregated in one

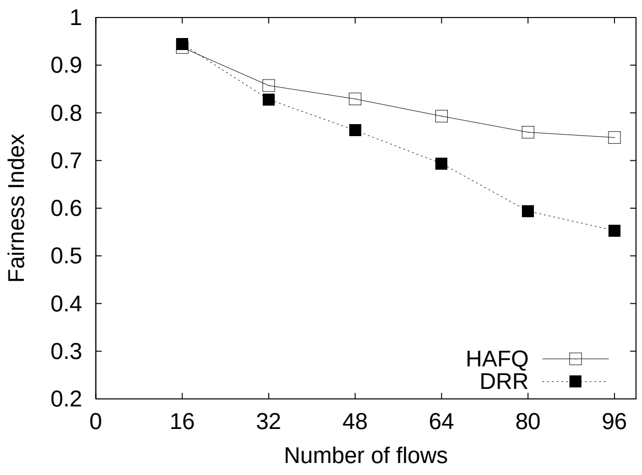


Figure 16: Fairness in the case of the different number of flows.

queue is small when the number of queues is large and the rate variability of each flow is small by improving an accuracy of the flow count estimation and performing packet–dropping using a packet counter value.

We next change the number of entries 2, 3, $\cdots$ 10 while the number of queues is fixed at eight. In this case, the required memory capacity is 128, 160, $\cdots$, 384. This is also because of improving an accuracy of the flow count estimation and performing packet–dropping.

In general, there is a difference in the memory capacity between edge routers and core routers. Therefore, a scalable packet scheduling scheme must be able to minimize the performance degradation as the memory capacity becomes small. Our scheme provides more excellent fairness than the DRR scheme as the memory capacity becomes limited. In other words, our scheme provides a more scalable packet scheduling than the DRR scheme.

4.3.4 Applicability to High–Speed Routers

The processing capacity of the IXP1200 is not high, and packet processing capacity is limited up to about 200 KPPS in our experiments. However, the router with a 10 Gbps line interface should have 100 times larger packet processing capacity than that of IXP1200. Therefore, we last discuss the capability of our scheme for 10 Gbps line interfaces.

Whether packet scheduling schemes can accommodate 10 Gbps line interfaces or not greatly depends on the processing capacity and memory access bandwidth of routers. With such high–

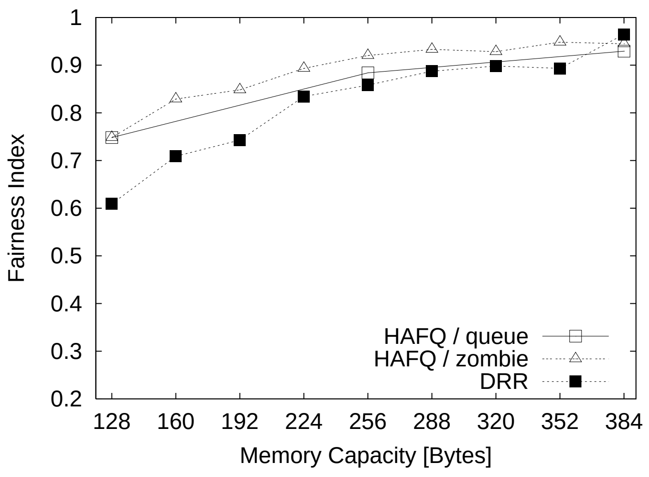


Figure 17: Fairness in the case of the different memory capacity

speed lines, routers can spend only 40 ns in processing one packet, and complex operations would easily lead to the performance degradation. However, the processing capacity has been rapidly improved and this problem can be expected to be solved in the near future. On the other hand, the memory access bandwidth seems to be an obstacle even in the future. Our scheme requires 13 memory accesses and the DRR scheme requires 7 memory accesses in packet input / output operations. Therefore, they require 325M and 175M memory accesses in 10 Gbps line interfaces. Thus, those schemes must not use an off–chip memory but an on–chip memory. If we use the on–chip memory, the memory access bandwidth is not a problem, but the memory capacity is very limited. However, even in such a circumstance, our scheme can accommodate many active flows in the practical number of flows keeping fairness on some level in case of the limited memory usage; for example, if our scheme has six entries in each zombie list and 1K queues, the required memory capacity is 32 Kbytes as can be estimated from Eq.(13). That is, our scheme can use an on–chip memory. On the other hand, the DRR scheme requires 6 Mbyte memory capacity for 192,000 queues for achieving the same fairness index.

5 Conclusion

The new scalable queue management scheme described in this thesis provides fair per–flow service in backbone networks. The scheme estimates the number of flows aggregated in a queue and allocates the bandwidth to the queue proportionally. It also improves fairness among flows in the same queue by preferentially discarding the packets of flows having higher arrival rates.

First, we have evaluated this scheme through simulation studies and compared it with two standard queue management schemes: FIFO and per–flow scheduling. We have shown that per–flow scheduling provides an excellent fairness only when all routers in the network is equipped with the number of queues and exactly one flow is accommodated in one queue; but it is impractical. Our new scheme provides a scalable packet scheduling mechanism and improves the fairness by the aggregated queuing with flow count estimation. We have shown that our new scheme is very effective in assuring the fairness of high–speed routers accommodating a large number of active flows.

Second, we have implemented our scheme on the IXP1200 network processor and evaluated scheduling complexity through our experimental measurements. Although our scheme requires additional instructions and memory accesses when compared with the DRR scheme, the performance degradation is minimal and our scheme provides a good per–flow fairness for all flows. Especially in high–speed routers where available memory capacity is extremely restricted in the current technologies, our scheme provides excellent fairness than the DRR scheme.

For future work, we will evaluate the performance of the HAFQ algorithm in the other situations. We also evaluate its performance by actual experiments.

Acknowledgements

I would like to express my sincere appreciation to Professor Hideo Miyahara of Osaka University, who introduced me to the area of computer networks including the subjects in this thesis.

All works of this thesis would not have been possible without the support of Professor Masayuki Murata of Osaka University. It gives me great pleasure to acknowledge his assistance. He has been constant sources of encouragement and advice through my studies and preparation of this manuscript.

I would like to express my great thankfulness to Dr. Hideyuki Shimonishi of Networking Research Laboratories, NEC corporation. All works of this thesis also would not have been possible without him.

I would like to express my unbounded gratitude to Mr. Akio Tomobe of Intel Corporation, who denotes the IXP1200 network prosessor, equipments of related to it, and so on.

Thanks are also due to Professor Shinji Shimojo of Osaka University who gave me many advices.

I am also indebted to Assistant Professor Naoki Wakamiya, Research Associates Hiroyuki Ohsaki, Go Hasegawa and Shin'ichi Arakawa of Osaka University who gave me helpful comments and feedbacks.

Finally, I thank many friends and colleagues in the Department of Informatics and Mathematical Science of Osaka University for their support.

References

[1] R. Mahajan and S. Floyd, “Controlling high bandwidth flows at the congested router,” tech. rep., International Computer Science Institute technical report TR-01-001, Apr. 2001.

[2] S. Floyd and K. Fall, “Promoting the use of end–to–end congestion control in the Internet,” *IEEE/ACM Transactions on Networking*, vol. 7, pp. 458–472, Aug. 1999.

[3] G. Hasegawa, K. Kurata, and M. Murata, “Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet,” in *Proceedings of IEEE International Conference on Network Protocol (ICNP) 2000*, pp. 177–186, Nov. 2000.

[4] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” *IEEE/ACM Transactions on Networking*, vol. 1, pp. 397–413, Aug. 1993.

[5] T. J. Ott, T. Lakshman, and L. Wong, “SRED: Stabilized RED,” in *Proceedings of IEEE INFOCOM 1999*, pp. 1346–1355, Mar. 1999.

[6] M. Shreedhar and G. Varghese, “Efficient fair queueing using deficit round robin,” *IEEE/ACM Transactions on Networking*, vol. 4, pp. 375–385, June 1996.

[7] D. Lin and R. Morris, “Dynamics of random early detection,” *ACM Computer Communication Review*, vol. 27, pp. 127–137, Oct. 1997.

[8] R. Kapoor, C. Casetti, and M. Gerla, “Core–stateless fair bandwidth allocation for TCP flows,” in *Proceedings of the IEEE International Conference on Communications (ICC) 2001*, June 2001.

[9] I. Stoica, S. Shenker, and H. Zhang, “Core–stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks,” in *Proceedings of the ACM SIGCOMM’98*, pp. 118–130, Oct. 1998.

[10] Z. Cao, Z. Wang, and E. Zegura, “Rainbow fair queueing: Fair bandwidth sharing without per–flow state,” in *Proceedings of IEEE INFOCOM 2000*, pp. 922–931, Mar. 2000.

[11] “Intel IXP1200.” available at `http://developer.intel.com/design/network/products/npfamily/ixp1200.htm`.

[12] Z. Cao, Z. Wang, and E. Zegura, "Performance of hashing–based schemes for Internet load balancing," in *Proceedings of IEEE INFOCOM 2000*, pp. 332–341, Mar. 2000.

[13] "UCB/LBNL/VINT network simulator - ns (version 2)." available at `http://www-mash.cs.berkeley.edu/ns/`.

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

[15] K. Claffy, H.-W. Braun, and G. Polyzos, "A parameterizable methodology for Internet traffic flow profiling," *IEEE Journals on Selected Areas in Communication*, vol. 13, pp. 1481–1494, Mar. 1995.

[16] "IXP1200 Developer Workbench." available at `http://www.intel.com/design/network/products/npfamily/sdk2.htm`.