Ordered Round-Robin: An Efficient Sequence Preserving Packet Scheduler

Jingnan Yao, Jiani Guo, *Member*, *IEEE*, and Laxmi Narayan Bhuyan, *Fellow*, *IEEE*

Abstract—With the advent of powerful network processors (NPs) in the market, many computation-intensive tasks such as routing table lookup, classification, IPSec, and multimedia transcoding can now be accomplished more easily in a router. An NP consists of a number of on-chip processors to carry out packet level parallel processing operations. Ensuring good load balancing among the processors increases throughput. However, such multiprocessing also gives rise to increased out-of-order departure of processed packets. In this paper, we first propose an Ordered Round-Robin (ORR) scheme to schedule packets in a heterogeneous NP assuming that the workload is perfectly divisible. The processed loads from the processors are ordered perfectly. We analyze the throughput and derive expressions for the batch size, scheduling time, and maximum number of schedulable processors. To effectively schedule variable length packets in an NP, we propose a Packetized ORR (P-ORR) scheme by applying a combination of deficit round-robin (DRR) and surplus round-robin (SRR) schemes. We extend the algorithm to handle multiple flows based on a fair scheduling of flows depending on their reservations. Extensive sensitivity results are provided through analysis and simulation to show that the proposed algorithms satisfy both the load balancing and in-order requirements for parallel packet processing.

Index Terms—Network processor, out of order, scheduling, round-robin.

1 Introduction

The next generation Internet is being designed to serve diverse content-aware applications instead of passively carrying packets from one node to another. These services demand service-differentiated and application-oriented processing at network nodes with quality of service (QoS) provision. Given the processing power limitation of single processor and the inherent parallelism associated with network traffic, parallel processing is becoming increasingly attractive.

Application-oriented services often carry stringent performance requirements, which are increasingly handled by multicore processors. As such, several companies have introduced powerful network processors (NPs) that can be placed in routers to execute various tasks in the network. These tasks can range from IP level table lookup algorithm to application level multimedia transcoding applications. Such an NP-based router permits sophisticated computations within the network by allowing their users to inject customized programs into the nodes of the network [1]. An NP provides the speed of an ASIC and at the same time is programmable. Each NP consists of a number of on-chip processors that can provide high throughput for network packet processing and application level tasks [2], [3], [4].

However, processing of packets belonging to the same flow by different processors gives rise to out-of-order departure of the packets from the NP and incurs high delay jitter for the outgoing traffic. For TCP, it has been proved that out-of-order transmission of packets is detrimental to the end-to-end performance [5]. Although TCP is designed to handle reordering, this involves additional processing at the TCP end points. TCP for Persistent Reordering (TCP-PR) [6] is a variant of TCP that attempts to improve TCP performance in the presence of persistent reordering phenomena. TCP-PR is a software-based solution and thus is restricted only at the end hosts. For many applications like multimedia transcoding and VoIP, it is imperative to minimize this out-of-order effect because the receiver may not be able to reorder them easily to tolerate high delay jitter. Therefore, efficient packet scheduling is necessary in order to guarantee both high throughput and minimal out-of-order delivery of packets. However, these two goals are contradictory to each other because scheduling on more processors improves throughput but also increases the out-of-order delivery of packets.

A plethora of scheduling schemes for multiprocessors has been proposed in parallel processing community. Examples of such schemes vary from simple static policies, such as round-robin (RR) or random distribution policy [1], [7], [8] to sophisticated adaptive load-sharing policies [9], [10], [11]. However, only simple policies such as RR are employed in practice because adaptive schemes are difficult to implement and involve considerable overhead. We have shown that RR is simple and fast but provides no guarantee to the playback quality of output streams of video signals because it causes large out-of-order departure of the processed media units [10]. Adaptive load-sharing scheme, which we

- *J. Yao is with the Wireless and Security Technical Group, Cisco Systems, 190 West Tasman Drive, Mail Stop SJCE/2/1, San Jose, CA 95134. E-mail: jinyao@cisco.com.*
- *J. Guo is with the Datacenter, Switching and Services Group, Cisco Systems, 210 West Tasman Drive, Mail Stop SJCF/1/4, San Jose, CA 95134. E-mail: jianiguo@cisco.com.*
- *L.N. Bhuyan is with the Computer Science and Engineering Department, University of California, Riverside, CA 92521-0304. E-mail: bhuyan@cs.ucr.edu.*

Manuscript received 14 Nov. 2007; revised 5 Apr. 2008; accepted 14 Apr. 2008; published online 20 May 2008.
Recommended for acceptance by A. Zomaya.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-11-0591.
Digital Object Identifier no. 10.1109/TC.2008.88.

implemented from the literature [9], achieves better unit order in output streams but involves higher overhead to map the media unit to an appropriate node.

Packet scheduling in an NP [9], [12], [13], [14] can be considered as similar to link aggregation techniques that employ multiple physical links from a source to the same destination. In fact, if the time for packet processing on a processor is replaced with an estimated transmission time of a packet, the two problems are equivalent. A practical link striping protocol, called surplus RR (SRR), is proposed by Adiseshu et al. [15] to schedule variable length packets over multiple links with different capacities. They also demonstrate that striping is equivalent to the classic load-balancing problem over multiple channels. They solve the variable packet size problem by transforming a class of fair queuing algorithms into load-sharing algorithms at the sender. Although their solution is elegant and efficient, it does not consider the transmission order among the packets in different channels. Hence, the receiver needs to run a resequencing algorithm to restore the packet order in the original flow. A strict synchronization between the sender and the receiver is difficult to implement. Cobb and Lin considered several sorting techniques in their paper [16] to avoid packet reordering that require access to upper layer protocol headers and thus potentially incur significant overhead. Moreover, the time to move packets between the single input/output port of the sender/receiver and different channels is assumed to be negligible in most of the channel striping papers. There should at least be a time overhead in executing the scheduler at the IP level processing, which is appropriately modeled in our scheme.

The aim of this paper is to derive an efficient packet-scheduling algorithm in an NP that comprises a number of processors for packet processing. It should provide 1) load balancing for processing variable length packets using a group of heterogeneous processors and 2) in-order delivery of packets without considering receiver's rearranging capability. We derive an Ordered RR (ORR) algorithm that considers a backlogged queue and can be adapted to dynamic arrival of packets. Like other packet scheduling algorithms such as RR/SRR, ORR is essentially a scheduler-based algorithm. Most of the sequence control schemes [17], [18] in the literature are based on maintaining extra sequence number or pointers in the packets. Our goal, however, is to minimize packet out of order without incurring extra overhead and without throughput degradation. All the control is at the scheduler and there is no extra mechanism or information maintained to keep the sequence.

The Divisible Load Theory (DLT) [19], [20] develops scheduling assuming that the workload is perfectly divisible for such distribution among the processors, so that all processors finish at the same time. It considers heterogeneous processors and different communication times to send data to those processors. DLT is particularly suitable for data parallel operations, where the volume of data can be perfectly distributed without causing any error. DLT is highly applicable in parallel processing of many applications [21]. In this paper, we demonstrate its possible use in an NP. As required for packet processing over multiple processors, DLT has to be tuned to consider sequential ordering of the packet transmission times. Our algorithm schedules the packets in batches by computing the minimal batch size, scheduling time, and number of schedulable processors given the maximum packet size and processing rates. A batch is similar to the concept of time epochs when scheduling is done. Several interesting results are derived regarding scalability of our algorithm.

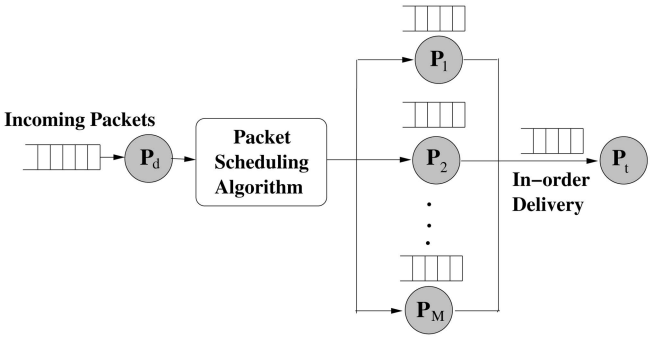


Fig. 1. Packet scheduling model for an NP.

Expressions for load distribution in heterogeneous processors are derived first by assuming that the schedulable workload is perfectly divisible in bytes. Since the arriving packets cannot be distributed to different processors in bytes, we derive a packetized version of the ORR algorithm by applying a combined version of deficit RR (DRR) and SRR algorithms [22], [15]. The Packetized ORR (P-ORR) algorithm produces better results in terms of throughput and out-of-order rate compared to RR and pure SRR schemes. To completely eliminate packet out of order, we propose a complement method to be used along with the P-ORR main algorithm. We then extend the P-ORR algorithm to handle multiple flows having reservations to satisfy both load balancing and fair scheduling requirements. We revise expressions of the batch size and packet dispatching condition to reflect the new requirement. Finally, we perform a number of simulations and sensitivity studies to verify the accuracy of our theory and obtain performance over wide-ranging input parameters.

The rest of this paper is organized as follows: In Section 2, we present the preliminaries and certain design issues in an NP. In Section 3, we design the ORR algorithm by giving theoretical derivation and analysis. In Section 4, we propose and design a packetized version of the ORR algorithm named Packetized-ORR (P-ORR) to deal with variable length packets. In Section 5, we present how to achieve fairness among multiple network flows using P-ORR. Simulation results are presented in Section 6 in comparison with several other schemes. Finally, in Section 7, we conclude this paper with future possible extensions related to this paper.

2 FIFO Packet Scheduling Formulation

Fig. 1 illustrates the multiprocessor architecture model of a router using an NP. The NP consists of one dispatching processor p_d, a few worker processors, p_1 through p_M, and a transmitting processor p_t. Intel IXP NP divides its set of microengines this way for packet processing [2]. The dispatching and transmitting processors communicate with

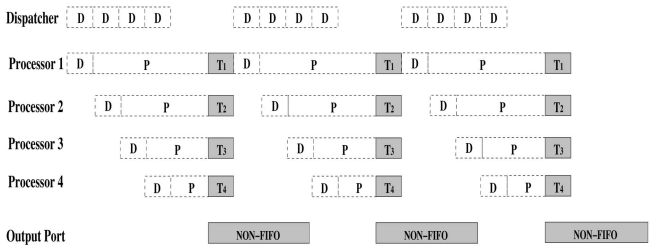


Fig. 2. Simultaneous completion pattern.

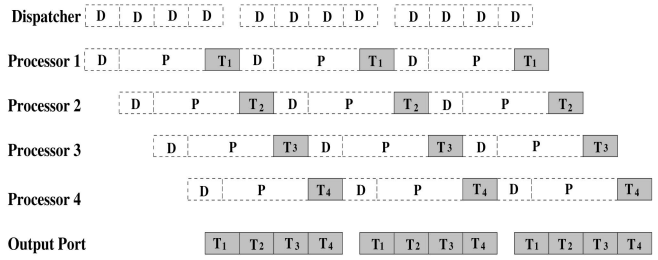


Fig. 3. Sequential completion pattern.

the I/O ports sequentially. The dispatching processor p_d implements the scheduling algorithm to split incoming traffic among the M worker processors for parallel packet processing. The transmitting processor p_t simply receives packets from processors p_1 to p_M on a First-Come-First-Serve (FCFS) basis and combines the traffic back into a single stream onto the output port. The aim of the packet scheduling algorithm is twofold: 1) the input load is balanced among the processors p_1 through p_M and 2) the flow order is maintained when the packets are transmitted to the transmitting processor p_t. In this paper, it is assumed that the packet processing time is proportional to the packet length. There are many scenarios, such as multimedia transcoding and crypto-based processing, which fall into this category. For others where the packet processing time is not directly related with packet size, some researchers have proposed processing time estimation or prediction mechanisms [23].

A similar problem called channel striping has been addressed in the literature by Adiseshu et al. [15]. There are N channels between the sender and the receiver. The sender implements the striping algorithm SRR to stripe incoming traffic across the N channels, and the receiver implements a resequencing algorithm to combine the traffic into a single stream. The striping algorithm aims to provide load sharing among multiple channels. It does not consider the transmission order among the packets in different channels. Hence, the receiver needs to run a resequencing algorithm to restore the packet order in the original flow. A strict synchronization between the sender and the receiver is difficult to implement.

Although the packet scheduling problem looks different from the channel striping problem, there are many similarities. First, in channel striping, the packets are transmitted in the channels. In NPs, the packets are processed on the worker processors. These two times are equivalent and proportional to the load size in bytes. Second, in channel striping, the time to move packets from the single input port to different channels is assumed to be negligible in [15]. Actually, there should be a time overhead in executing SRR at the IP level processing. As for packet scheduling in an NP, the time to move packets from the dispatching processor to a worker processor cannot be ignored because of the time taken by the dispatching processor and the transmission between the two processors. Finally, the transmitting processor in an NP removes the packets from the worker processors on an FCFS basis, whereas the receiving processor in the striping model executes a resequencing algorithm. Hence, the models developed in this paper are applicable both to packet processing in an NP or packet transmission over multiple channels.

For the rest of this paper, we will explain our models/algorithms just in terms of NPs without loss of generality. To describe the whole procedure experienced by one packet when it is processed in an NP, there are three steps: 1) D-step: the dispatching processor dispatches the packet to a worker processor, 2) P-step: the worker processor processes the packet, and 3) T-step: the worker processor sends the packet to the transmitting processor. Correspondingly, in the channel striping problem, only one step, namely P-step, is modeled, if we consider transmission of a packet as a type of processing. Hence, the packet scheduling problem in an NP is more complicated.

We start by exploring how DLT works for scheduling the traffic. As suggested by the DLT literature [20], in order to obtain minimal processing time, it is *necessary and sufficient* that all the participating processors finish processing at the same time instant. This condition is referred to as an *optimality principle* and serves as a baseline of the load distribution strategies. In this way, the traffic is well balanced across the processors and the communication bandwidth is fully utilized. However, if we take a look at how such a scheme works by observing the packets at the output port, some problems are evident. As shown in Fig. 2, all processors complete processing at the same time and thus intend to deliver packets to the transmitting processor simultaneously. Such a simultaneous completion pattern causes concurrent data delivery at the output port. As a result, uncontrollable interleaving among packets is incurred and the problem deteriorates as the number of worker processors grows.

Thus, instead of letting all the processors complete processing simultaneously, we propose a sequential completion pattern for packet transmission to ensure in-order data delivery at the output port, as illustrated in Fig. 3. Let there be M worker processors, each processor p_i, $\forall i$, first receives some packets from the dispatching processor p_d (D-step), then processes these packets (P-step), and finally sends the packets to the transmitting processor p_t (T-step) sequentially. We propose to let the dispatching processor distribute the packets among the M worker processors from p_1 to p_M in such a way that each processor completes processing sequentially. In another word, the worker processor p_i for $i = 2 \sim \mathrm{M}$ starts delivering the packets to the p_t immediately after p_{i-1} completes its T-step. Note that such a sequential completion pattern for packet processing ensures in-order packet delivery at the output port; however, the actual processing on different processings is essentially in parallel. In this way, packet sequence is

TABLE 1
Notations and Terminology

Symbol	Definition of the Symbol
M	The total number of worker processors.
p_i	The ith worker processor, $1 \leq i \leq M$.
w_i	The processing rate of p_i measured in sec/byte.
$z_{r,i}$	Bandwidth of link $l_{r,i}$ measured in sec/byte.
$z_{s,i}$	Bandwidth of link $l_{s,i}$ measured in sec/byte.
N	The number of active incoming flows.
f_j	The jth traffic flow, $1 \leq j \leq N$.
r_j	The reservation of flow f_j, $0 \leq r_j \leq 1$.
I	The minimal number of bytes to be scheduled in a batch.
L	The maximal possible packet length that may arrive.
B	The batch size, i.e., the number of bytes that is to be scheduled in a round. Technically, B can be any value larger than I. In this paper, it is defined as a multiple of the minimal schedulable batch size, i.e., $B = mI$, where m is a positive integer.
m	The batch granularity, which represents the ratio of B/I. m is a positive integer. The bigger m is, the coarser the batch granularity.
$M_{saturate}$	Assuming homogeneous NP configuration, $M_{saturate}$ is the maximal number of worker processors that can be supported by one dispatching processor.
α	This refers to a load distribution of the input load, i.e., $\alpha = (\alpha_1, \ldots, \alpha_i, \ldots, \alpha_M)$, where α_i is the fraction of load assigned to p_i such that $0 < \alpha_i < 1$ and $\sum_{i=1}^{M} \alpha_i = 1$.
Q_i	Given the batch size B for a scheduling round, Q_i is the quantum for processor p_i. That is, $Q_i = \alpha_i B$ and $\sum_{i=1}^{M} Q_i = B$.
F_j	Given the batch size B for a scheduling round, F_j is the quantum for flow f_j. Note that $F_j = r_j * B$ and $\sum_{j=1}^{N} F_j = B$.
D_i	Packet dispatching phase of a scheduling round. Thus, it is defined as the time duration p_d takes to dispatch B_i to p_i, i.e., $D_i = \alpha_i B z_{r,i}$.
P_i	Packet processing phase of a scheduling round. Time duration p_i takes to process Q_i, i.e., $P_i = \alpha_i B w_i$.
T_i	Packet receiving phase of a scheduling round. Time duration p_i takes to send Q_i to p_t, i.e., $T_i = \alpha_i B z_{s,i}$.
Gap_d	Idle period on p_d between two batches.
Gap_i	Idle period on processor p_i between two batches.
Gap_t	Idle period on p_t between two batches.

maintained at the output port, as well as the load is balanced among multiple processors. We name this scheduling algorithm ORR as it dispatches the packets in an RR manner while ensuring packet order. A nice property of the above model is that sequential data delivery is achieved without any additional control. Simply by arranging the dispatching and processing phases, a scheduling algorithm is obtained.

To design a practical algorithm, a number of issues need to be addressed. First, how many packets should be dispatched to each processor to produce the desired pattern? Second, what if the packets are of variable lengths? Third, given any NP configuration and packet arrival pattern, can we always find a way to schedule packets in such a sequential delivery pattern? Fourth, how can we ensure fair scheduling among multiple flows having different reservations? The rest of this paper attempt to analyze the situations and provide answers to the above questions.

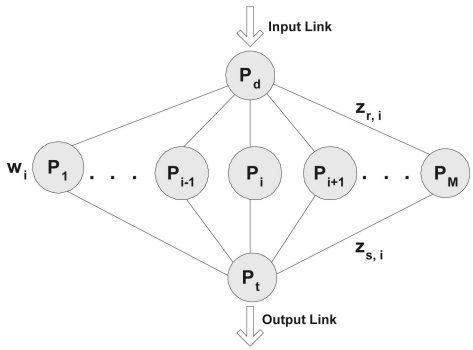


Fig. 4. Mathematical model.

Table 1 defines the notations that are used in this paper. First, we devise an ORR algorithm assuming that the workload is perfectly divisible. In the theoretical approach, a load distribution is first determined to produce the sequential scheduling pattern in one scheduling round. To further enable dynamic scheduling, we derive expressions for the *minimal batch size* and the *batch size* based on the maximal possible packet length. Then, the complete ORR algorithm is designed to schedule packets over multiple scheduling rounds taking extra care to ensure the sequential delivery of multiple batches of data. The scalability of the ORR algorithm is analyzed and important conclusions are reached. To schedule variable length packets, a packetized version of the ORR algorithm (P-ORR) is devised based on the combination of DRR and SRR. In this way, we always minimize the absolute difference between the actual load distribution and the ideal load distribution. Finally, the P-ORR algorithm is extended to handle multiple flows having reservations.

3 Ordered Round Robin

In this section, we formulate a theoretical model for the general packet scheduling problem in an NP and develop a load scheduling algorithm ORR to produce a scheduling pattern described in Fig. 3. There are two design goals for ORR. The first is to ensure load balancing for a group of heterogeneous processors. The second is to ensure strict in-order delivery of packets. In the following derivation, we assume that the input load is divisible at the granularity of 1 byte.

3.1 Optimal Load Distribution for a Single Round

For an NP, we set up the following mathematical model for ease of theoretical analysis. As shown in Fig. 4, there are $(M+2)$ processors and $2M$ links inside the router. The worker processors $p_1, p_2, \ldots, p_M$ are connected to the dispatching processor p_d via links $l_{r,1}, l_{r,2}, \ldots, l_{r,M}$. Meanwhile, each worker processor has a direct link $l_{s,i}$ to the transmitting processor p_t. The dispatching processor p_d receives packets from the input link, divides the input load into M parts, and then distributes these load fractions to the corresponding worker processor. Each worker processor p_i, upon receiving its load fraction α_i, starts processing immediately and continues to do so until this fraction is finished. Finally,

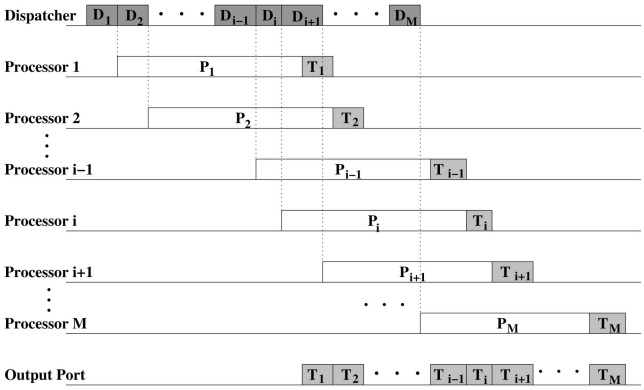


Fig. 5. Load distribution in a single round.

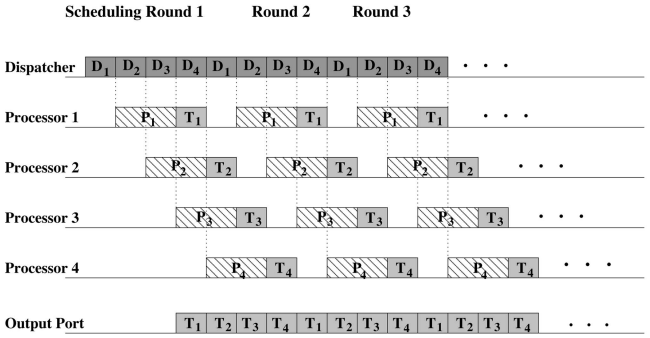


Fig. 6. Optimal load distribution in multiple rounds.

p_i sends the processed fraction to the transmitting processor p_t via link $l_{s,i}$. Note that α_i is the fraction of load assigned to p_i such that $0 \le \alpha_i \le 1$ and $\sum_{i=1}^{M} \alpha_i = 1$. For simplicity of the equations, all the above rates are defined with a dimension of seconds per byte, which is the inverse of the actual communication/processing bandwidth.

As illustrated in Fig. 5, the load is partitioned among processors such that all the worker processors stop processing sequentially and, therefore, deliver the processed fractions to the transmitting processor sequentially. That is, worker processor p_{i+1} starts delivering the packets to the output port only after and right after p_i completes its delivery. All the packets are processed in parallel and are sent to the output port in order without any break. In this way, we eliminate the out-of-order packet delivery. To achieve such a sequential delivery pattern, we obtain the following recursive equations for $i = 1, \ldots, M-1$ from the timing diagram:

$$P_i + T_i = D_{i+1} + P_{i+1}. \tag{1}$$

As defined in Table 1, D_i, P_i, and T_i are time durations for p_d to dispatch Q_i onto p_i, for p_i to process Q_i, and for p_i to send Q_i to p_t, respectively. That is, $D_i = \alpha_i B z_{r,i}$, $P_i = \alpha_i B w_i$, and $T_i = \alpha_i B z_{s,i}$ assuming a batch size B. These recursive equations can be solved by expressing all the α_i $(i = 1, \ldots, M-1)$ in terms of α_M and using the normalizing equation $\sum_{i=1}^{M} \alpha_i = 1$, i.e.,

$$\alpha_i = \alpha_M \prod_{v=i}^{M-1} \Phi_v, \ i = 1, \ldots, M, \tag{2}$$

where

$$\alpha_M = \frac{1}{1 + \sum_{u=1}^{M-1} \prod_{v=u}^{M-1} \Phi_v}, \ \Phi_v = \frac{z_{r,v+1} + w_{v+1}}{w_v + z_{s,v}}. \tag{3}$$

3.2 Batch Size Determination

As presented in Section 3.1, we can establish a partition of any given load S, calculated as $(Q_1, Q_2, \ldots, Q_M)$ and $Q_i = \alpha_i S$ $(i = 1, 2, \ldots, M)$, to ensure both load balancing and sequential data delivery in a single scheduling round. In this section, we design a dynamic packet scheduling algorithm for NPs based on this observation. For an NP to schedule packets, it repeats the sequential scheduling pattern over time while packets arrive dynamically, as illustrated in Fig. 6. Within each round, load balancing and sequential delivery are ensured for a batch of packets. To design such a scheduling algorithm, we first need to determine a feasible batch size for each scheduling round, i.e., the number of packets to be scheduled in each round.

To find a feasible batch size, we note that at least one packet should be dispatched to each processor in a scheduling round. Therefore, we define an important system parameter *minimal schedulable batch size* I as the total bytes that should be scheduled in one scheduling round. Suppose the maximal possible length of a packet (in terms of bytes) that may arrive at the NP is L, the minimal schedulable batch size I is defined to be

$$I = CL, \tag{4}$$

where C is the minimal positive integer such that at least one packet fits into the load fraction that can be dispatched to a worker processor, i.e.,

$$C = \left\lceil \frac{1}{\min_i(\alpha_i)} \right\rceil. \tag{5}$$

Hence, CL constitutes a minimal batch size that should be guaranteed for one scheduling round. Given I as the minimal batch size, the *batch size* B is set to be a multiple of I as follows:

$$B = mI, \tag{6}$$

where m is a positive integer referred to as *batch granularity*. In the following section, we will see how the batch granularity m affects the system throughput given the minimal schedulable batch size I.

Now, we can do dynamic scheduling as follows: The load fraction Q_i that should be dispatched to the ith worker processor p_i is calculated as $Q_i = \alpha_i B$. During any backlogged interval, the scheduling and the desired delivery pattern will not be interrupted. So long as the packets arrive at a rate such that there are always Q_i bytes to be scheduled to the current chosen processor p_i, the scheduling can run smoothly. In the following discussion, we assume a continuously backlogged system for ease of discussion.

3.3 Scheduling Time Determination

In this section, we determine the best scheduling time of each round. There are two questions to be answered. First, is there

any resource conflict between two adjacent scheduling rounds if we schedule multiple rounds continuously? Second, how can we resolve the potential resource conflict if it arises?

3.3.1 Optimality Analysis

For an optimal solution, we shall partition the load such that there is no idle time for any of the processors in the router while still maintaining a continuous sequential delivery pattern for the passing streams. To illustrate these ideas, we show the desired timing diagram in Fig. 6 for a load distribution of M worker processors and R rounds. As can be observed from the diagram, multiple batches have been joined together seamlessly to carry out the entire transcoding task. There is no idle time on both receiving processor and any of the worker processors at any time instant. The entire load is therefore divided into $(M \times R)$ parts such that the $(j+1)\text{th}$ batch arrives at a worker processor right after it finishes delivering the current batch to the output port. In this way, all the processed packets are timely and sequentially sent to the output link, and none of the processors is idle. To determine such an optimal load distribution, we can observe the following recursive equations from the timing diagram:

$$\text{for } (i = 1 \sim M-1, j = 1 \sim R)$$

$$\alpha_{i,j} w_i + \alpha_{i,j} z_{s,i} = \alpha_{i+1,j} z_{r,i} + \alpha_{i+1,j} w_{i+1} \tag{1-1}$$

$$\text{for } (j = 1 \sim R-1)$$

$$\alpha_{M,j} w_M + \alpha_{M,j} z_{s,M} = \alpha_{1,j+1} z_{r,1} + \alpha_{1,j+1} w_1 \tag{1-2}$$

$$\text{for } (i = 2 \sim M, j = 1 \sim R-1)$$

$$\alpha_{i,j} z_{s,i} = \alpha_{i-1,j+1} z_{r,i-1} \tag{2-1}$$

$$\text{for } (j = 1 \sim R)$$

$$\alpha_{1,j} z_{s,1} = \alpha_{M,j} z_{r,M}. \tag{2-2}$$

The above two sets of equations are independent and can be solved uniquely by expressing all the $\alpha_{i,j}$ in terms of $\alpha_{M,R}$ and using the normalizing equation $\sum_{j=1}^{R} \sum_{i=1}^{M} \alpha_{i,j} = 1$. Thus, it is unlikely to obtain an α satisfying both the equations unless the system parameters are constrained by certain relationship. For a homogeneous network with $w_i = w$, $z_{r,i} = z_r$, and $z_{s,i} = z_s$, we derive the following constraint for the optimal solution:

$$\frac{z_r}{z_s} = \left(\frac{z_r + w}{w + z_s}\right)^{M-1}, \tag{7}$$

which holds true if $z_r = z_s$.

We found that the practical implementation may not necessarily satisfy the optimality requirement of (7). Thus, there is no general optimal solution for a generic router configuration. The nonexistence of such a general optimal solution is due to the fact that we want to maintain the flow order of the incoming packets to eliminate the out-of-order problem and at the same time best utilize the computing power of the processors. This incurs two sets of independent equations when we are to determine α, thus inevitably adding constraints on the system parameters to satisfy both of the design goals. Therefore, to achieve a feasible solution for a general NP setting, somehow we need to relax one of these constraints in our equations. Based on this observation, we propose a relaxed solution with scheduling gaps introduced as follows.

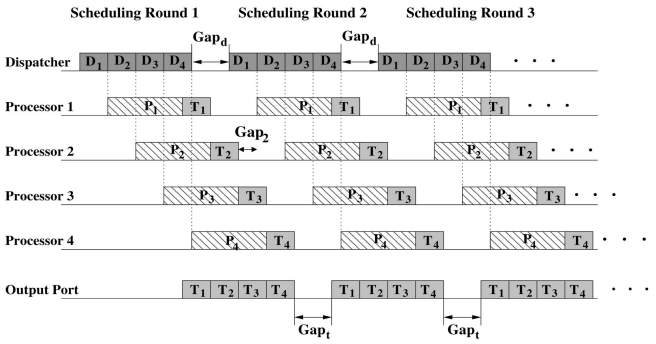


Fig. 7. Gapped load distribution in multiple rounds.

3.3.2 Gapped Solution

Note that the basic requirement of a sequential delivery pattern that consists of multiple rounds is that the scheduling of two adjacent batches cannot be overlapped to avoid resource conflict. For each processor, the D-step cannot be started before the T-step of the previous round completes. For example, as shown in Fig. 7, D_4 completes while processor p_1 is still in its T_1 phase; therefore, D_1 for the second round cannot be started and an idle period Gap_d is introduced on p_d to avoid resource conflict on processor p_1. Similarly, the transmitting processor p_t may also see a gap, Gap_t, if the receiving of T_1 does not follow the receiving of T_4 in the previous round immediately, as shown in Fig. 7. In addition, gaps may be observed between the processing of two adjacent batches on a worker processor, because the processor needs to wait for p_d to start the dispatching of the next round. We denote this gap on the $i\text{th}$ processor p_i as Gap_i.

Note that the FIFO queues at each processor are used for rate adaptation from/to the dispatching and transmitting processor. The FIFO is assumed to have limited depth, e.g., one batch size of a certain processor. The existence of such FIFOs does not eliminate the requirement that gaps may be introduced between adjacent scheduling round, because in a situation when all processors' aggregated processing capacity cannot keep up with the dispatching/transmitting data rate, the FIFOs will overflow eventually if no gap exists. Assume that the batch size is uniform over all scheduling rounds, from the timing diagram, the following equations are obtained for a heterogeneous system:

$$\begin{aligned} Gap_i &= \left(\Sigma_{k=1}^{M} D_k + Gap_s + \Sigma_{k=1}^{i-1} D_k\right) - \left(\Sigma_{k=1}^{i} D_k + P_i + T_i\right) \\ &= \Sigma_{k=1}^{M} D_k + Gap_s - (D_i + P_i + T_i), \end{aligned} \tag{8}$$

$$\begin{aligned} Gap_r &= \left(\Sigma_{k=1}^{M} D_k + Gap_s + D_1 + T_1\right) - \left(\Sigma_{k=1}^{M} T_k + T_M + R_M\right) \\ &= \Sigma_{k=1}^{M} D_k + Gap_s - \Sigma_{k=1}^{M} R_k. \end{aligned} \tag{9}$$

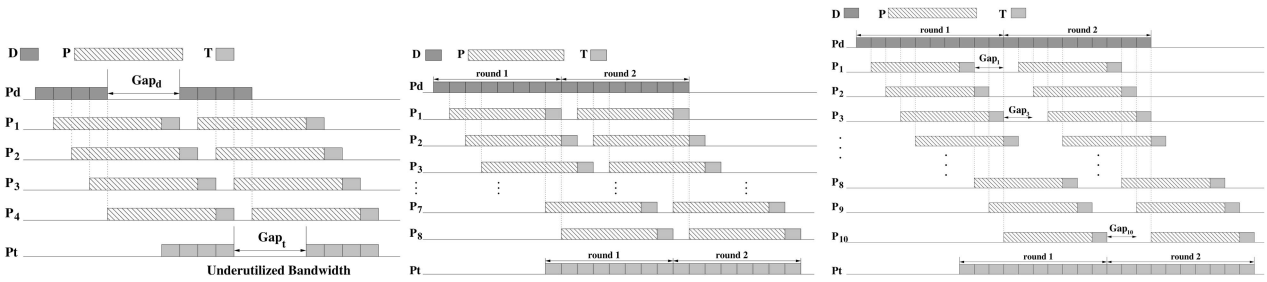


Fig. 8. Scheduling time determination in a homogeneous system. (a) $M = 4,\ Gap_d = Gap_t > 0,\ Gap_i = 0$. (b) $M = 8,\ Gap_d = Gap_t = Gap_i = 0$. (c) $M = 10,\ Gap_i > 0,\ Gap_d = Gap_t = 0$.

For a feasible scheduling pattern, none of Gap_d, Gap_t, and Gap_i $(i = 1, 2, \ldots, M)$ can be negative to avoid resource conflicts. Essentially, our goal is to find out the minimum Gap_d such that Gap_t and Gap_i are nonnegative. Based on this rationale, we can derive the following constraints:

$$\begin{aligned} Gap_i \geq 0 &\Leftrightarrow \Sigma_{k=1}^{M} D_k + Gap_d - (D_i + P_i + T_i) \geq 0 \\ &\Leftrightarrow Gap_d \geq max_i(D_i + P_i + T_i) - \Sigma_{k=1}^{M} D_k, \end{aligned} \tag{10}$$

$$\begin{aligned} Gap_t \geq 0 &\Leftrightarrow \Sigma_{k=1}^{M} D_k + Gap_d - \Sigma_{k=1}^{M} T_k \geq 0 \\ &\Leftrightarrow Gap_d \geq P_M + T_M - D_1 - P_1. \end{aligned} \tag{11}$$

Thus, from (10) and (11), we have

$$Gap_d^{min} = \max\Big(\big(\max_i(D_i + P_i + T_i) - \Sigma_{k=1}^{M} D_k\big), (P_M + T_M - D_1 - P_1)\Big), \tag{12}$$

$$Gap_i^{min} = Gap_d^{min} + \Sigma_{k=1}^{M} D_k - (D_i + P_i + T_i), \tag{13}$$

$$Gap_t^{min} = Gap_d^{min} + \Sigma_{k=1}^{M} D_k - \Sigma_{k=1}^{M} T_k. \tag{14}$$

Hence, in a heterogeneous system, as long as the dispatching processor maintains a gap Gap_d^{min} between two adjacent batches, a scheduling algorithm that ensures both sequential delivery and load balancing is obtained.

In a homogeneous system, we denote $z_{r,i} = z_{s,i} = z$ and $w_i = w$ for $i = 1, 2, \ldots, M$. According to (2) and (3), $\alpha_i = 1/M$ for $i = 1, 2, \ldots, M$ in a homogeneous system. According to (4) and (5), the minimal schedulable batch size is calculated to be $I = ML$. The batch size B is then set to be $B = mI = mML$, where m is a positive integer. Hence, the load dispatched to a processor in a single batch can be uniformly denoted as $B/M = mL$. Therefore, for a homogeneous system, (12), (13), and (14) can be simplified as

$$Gap_d^{min} = max((w + 2z - zM)mL, 0), \tag{15}$$

$$Gap_i^{min} = max\big(Gap_d^{min} + (zM - w - 2z)mL, 0\big), \tag{16}$$

$$Gap_t^{min} = Gap_d^{min}. \tag{17}$$

Equations (15), (16), and (17) suggest that

$$Gap_d = Gap_t = \begin{cases} (w + 2z - zM)mL, & M < M_{saturate}, \\ 0, & M \geq M_{saturate}, \end{cases}$$

$$Gap_i = \begin{cases} (zM - w - 2z)mL, & M > M_{saturate}, \\ 0, & M \leq M_{saturate}, \end{cases}$$

where $M_{saturate}$ is

$$M_{saturate} = \lceil w/z + 2 \rceil. \tag{18}$$

Hence, it can be concluded that $M_{saturate}$ is solely determined by the processing rate w and the packet dispatching/transmitting rate z at p_d/p_t. All gaps are eliminated when $M_{saturate}$ worker processors are adopted, hence the maximum throughput.

3.4 Scalability Analysis

Consider a homogeneous system with M identical processors. According to the analysis presented in the above section, the best scheduling time is determined by the number of worker processors M, the maximal packet length L, and the batch granularity m. $M_{saturate}$ is calculated according to the system parameters (w, z) as $M_{saturate} = w/z + 2$. $M_{saturate}$ actually represents the optimal number of worker processors that the dispatching and transmitting processor can support. Fig. 8 demonstrates such a scheduling example, where $w = 6\ \mu$s/byte and $z = 1\ \mu$s/byte, and hence, $M_{saturate} = 8$.

Very interesting conclusions about the system scalability have thus been reached. 1) If $M < M_{saturate}$, as shown in Fig. 8a, a gap should be introduced between the initiation of two adjacent batches at the dispatching processor p_d. The length of this gap is defined as Gap_d in (15). In this case, the processing rate of worker processors cannot match the dispatching and transmitting rate and thus causes p_d and p_t to wait for the worker processors to complete processing. 2) If M is equal to $M_{saturate}$, adjacent scheduling rounds are initiated continuously without introducing gaps in between, as shown in Fig. 8b. All processors are fully utilized and there is no break between the delivery of adjacent batches. In this case, the dispatching/transmitting rate and the processing rate match perfectly to give the best performance. 3) If $M > M_{saturate}$, no gap should be introduced at p_d between two adjacent scheduling rounds although idle time Gap_i starts to appear on individual processors p_i, as

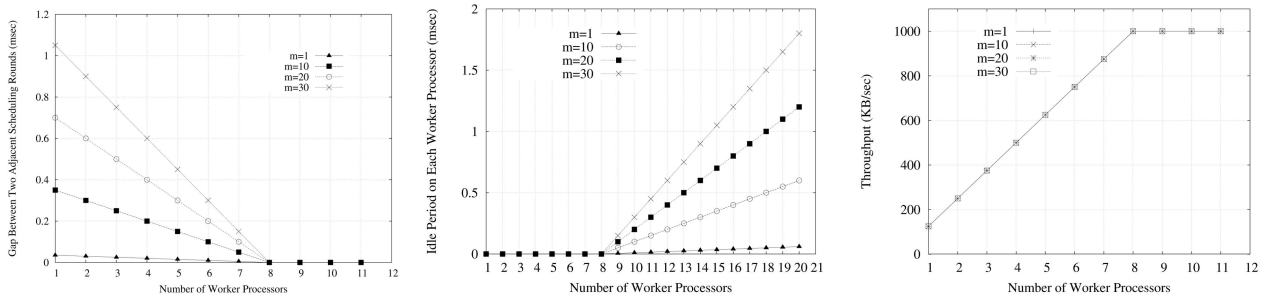


Fig. 9. Scalability analysis. (a) Gap_d and Gap_t between two adjacent scheduling rounds. (b) Gap_i on each worker processor. (c) Scalability of the throughput.

shown in Fig. 8c. In this case, the dispatching and transmitting rate of p_d and p_t cannot cope with the processing rate of the processors. The processing power of the processors is underutilized because the communication overhead at the dispatching/transmitting processor becomes the bottleneck.

Note that $M_{saturate}$ is determined only by the ratio of w/z and, therefore, independent of the batch size, where w is the processing rate of a processor and z is the packet dispatching/transmitting rate at p_d/p_t. Hence, for a given NP system with configuration parameters defined as (w, z) and assuming that processing always consumes greater time than the dispatching and transmitting, i.e., $w > z$, there always exists an optimal number of worker processors that one pair of dispatching/transmitting processor can support.

Fig. 9 presents the gaps and throughput for different values of m (batch granularity) as a function of the number of processors. In addition to the parameters mentioned earlier, the maximal packet length is now fixed as $L = 5$ Kbytes. For the homogeneous case, $Gap_d = Gap_t$ and Gap_i are same for all worker processors. Fig. 9a shows that Gap_d and Gap_t decrease linearly as the number of processors increases until $M = 8$. For a given number of processors, Gap_d and Gap_t increase linearly with increasing batch granularity m. After the system saturates at $M = 8$, the dispatching/transmitting processor hits the bottleneck point where $Gap_d = Gap_t = 0$, thus it cannot supply any more packets to the processors even if they are free.

Fig. 9b shows that the worker processors are all busy until $M = 8$. Then, gaps appear and they become idle for certain time depending on the batch granularity m, where they become underutilized because the dispatching processor has reached the bottleneck point. Also, note that as we increase batch granularity m, the idle period on worker processors (Gap_i) increases. Clearly, all gaps are minimized when the minimal batch size I is adopted, i.e., $m = 1$.

Fig. 9c shows the variation in system throughput (in terms of kilobytes per second) as a function of the number of worker processors M and batch granularity m. $M_{saturate}$ represents the saturating point of the system throughput. The system throughput is flat after the number of worker processors exceeds $M_{saturate}$. This is a result of underutilization of the processors, as illustrated in Fig. 9b. In addition, the system throughput is independent of the batch granularity m. After the number of worker processors reaches the saturation point $M_{saturate}$, the dispatching processor achieves zero idle time and keeps pumping traffic into the system, and therefore, the throughput stays at p_d's dispatching rate.

4 Packetized Ordered Round Robin

In the previous section, a dynamic scheduling algorithm ORR has been designed for an NP system to ensure both load balancing and in-order delivery assuming that the load is divisible at a granularity of 1 byte. In practice, we have to schedule workload at the granularity of one packet, which may be of variable length. To schedule variable length packets, we design a packetized version of ORR (P-ORR) in this section.

According to the ORR algorithm, given a batch size B, the ideal load distribution among multiple processors in any scheduling round is calculated as $(Q_1, Q_2, \ldots, Q_M)$, where $Q_i = \alpha_i B$ for $i = 1, 2, \ldots, M$. Because the workload contains variable length packets, the actual load distribution $(\tilde{Q}_1, \tilde{Q}_2, \ldots, \tilde{Q}_M)$ may deviate from its ideal amount. To guarantee the desired sequential delivery pattern, the actual load distribution should match the theoretical value as close as possible. *Therefore, the design goal of the P-ORR algorithm is to minimize the discrepancy* $\sum_{i=1}^{M}(\tilde{Q}_i - Q_i)^2$.

The proposed P-ORR algorithm is based on a combination of the SRR [15] and DRR [22] schemes. The packet scheduler at the dispatching processor first calculates $(Q_1, Q_2, \ldots, Q_M)$ to determine the ideal number of bytes that should be dispatched on each worker processor in a scheduling round. It also uses a pointer, denoted as $which$, to point to the processor that actively receives packets. So, Q_{which} represents the ideal number of bytes that should be dispatched to the processor p_{which}. As the scheduling proceeds, the size of the packets that have been dispatched to p_{which} is deducted from Q_{which}. The remaining load is recorded in a variable $Balance_{which}$. Ideally, the packet scheduler will dispatch a maximum of $Balance_{which}$ bytes to the processor p_{which} before it changes the pointer $which$ to point to the next processor. However, because of the variable packet length, it may not be possible for a packet to exactly fit into $Balance_{which}$. If a packet size exceeds $Balance_{which}$, the scheduler must decide as to dispatch the packet or not. The dispatching decision is made as follows: Let the packet size be $PSize$, if $PSize < Balance_{which}$, the

packet is scheduled to the processor p_{which}. Otherwise, it is scheduled only if

$$(PSize - Balance_{which}) \leq Balance_{which}. \quad (19)$$

The rationale behind the algorithm is that: in (19), the left-hand side represents the absolute value of $|Balance_{which}|$ when the packet is scheduled; and the right-hand side represents the absolute value of $|Balance_{which}|$ when the packet is not scheduled. We always make a decision to minimize the absolute value of $Balance_{which}$. This is because $|Balance_{which}| = |\tilde{Q}_i - Q_i|$, the absolute difference between the actual scheduled load and the ideal load, measured as $\sum_{i=1}^{M}(\tilde{Q}_i - Q_i)^2$, is also minimized. Note that (19) can be simplified as $Balance_i \geq PSize/2$.

While comparing with SRR [15] and DRR [22], our scheme improves over both by taking their combination. In each scheduling round, the absolute difference between the actual scheduled load and the ideal load, denoted as $|\tilde{Q}_i - Q_i|$, is bounded by the maximal packet length L in either DRR or SRR scheme; while in our proposed scheme, it is bounded by half of the maximal packet length $L/2$.

In the above scheme, we have minimized $\sum_{i=1}^{M}(\tilde{Q}_i - Q_i)^2$ for a single scheduling round. Over multiple scheduling rounds, the deviation of the actual load from the ideal load may be accumulated to be bigger and bigger. To avoid this, the ideal load Q_{which} for the next scheduling round is always adjusted as $Q_{which} = Q_{which} + Balance_{which}$ at the end of each scheduling round. The proposed P-ORR algorithm is presented in Fig. 10. It not only achieves load balancing in the presence of variable length packets but also ensures the minimal out-of-order transmission of the packets.

Under the observation that P-ORR always schedule packets sequentially from processor p_1 to processor p_M within a single round, we propose a simple but effective sequence control scheme to complement the P-ORR algorithm to reinforce packet ordering. The scheme requires that the dispatching processor p_d maintains an N-bit round ID, initially set to 0, and after each round of scheduling, the round ID is incremented by 1 (modulo 2^N). Each packet scheduled at p_d should have the round ID attached to it. Another round ID is maintained on the transmitting processor p_t side and set to 0 at system initialization. Upon the beginning of each round, p_t starts retrieving packets from processor p_1's FIFO that have the same round ID attached as the p_t's round ID. Once a mismatch happens, p_t will turn to processor p_2 and repeat the same process until processor p_M's packets are collected. The transmitting processor round ID will then be incremented by 1 (modulo 2^N) and starts the next round. Compared with the logical reception method, even with a single bit round ID ($N = 1$), our scheme is immune to single packet loss if we set the batch size greater than $2 * Max(Psize)$. Actually, our scheme can tolerate $(2^N - 2)$ consecutive batches of packet loss without losing the packet order at the transmitting processor. There are other sequence number-based schemes, e.g., total order sequencing and SCIMA+AFR schemes as evaluated in [18]. The total order sequencing method attaches a distinct and incremental sequence number to each packet across all processors, and SCIMA+AFR scheme attaches an incremental sequence number per processor and a next-processor pointer to each packet for packet loss detection and reordering purpose. Given the same packet loss tolerance requirement, our scheme has a lower cost (size of N) because the same round ID is attached to all packets in the same round. Furthermore, using larger batch size also enhances packet loss resilience.

Packetized Ordered Round Robin

Parameter Initialization:

(1) Using (2) and (3), determine $\alpha_i, \ \forall i$.
(2) Using (4) and (5), determine the minimal batch size.
(3) Set the batch size B as a multiple of I. Calculate the number of bytes that should be scheduled to each worker processor in a scheduling round. $Q_i = \alpha_i B, \ \forall i$.
(4) Using (12), calculate the gap Gap_d that should be adopted between the initiation of two adjacent batches.

Packet Scheduling:

```
which ← 1
Balance_which ← Q_which
while ( true )
    if (packet available)
        PSize ← size of the head-of-line packet
        if (Balance_which ≥ PSize/2)
            Dispatch the head-of-line packet to p_which
            Balance_which ← Balance_which − PSize
        else
            Q_which ← Q_which + Balance_which
            if ( which == M )
                    which ← 1
                    Balance_which ← Q_which
                    wait for Gap_d seconds
            else
                    which ← which + 1
                    Balance_which ← Q_which
    else   /* Non-backlog processing */
        Q_i ← α_i B, ∀i
        which ← 1
        Balance_which ← Q_which
        wait for Gap_d seconds
```

Fig. 10. P-ORR.

5 Fair Scheduling among Multiple Flows

In Section 4, a P-ORR algorithm is developed to schedule incoming packets among multiple processors. In the above discussion, we have assumed that the incoming packets are all treated equally. However, in practice, packets may belong to different network flows that have made different reservations. In this section, we extend the P-ORR algorithm to provide fair scheduling among multiple flows. We assume that all flows are continuously backlogged.

Let there be N flows, flow f_j has made a reservation r_j and the equation $\sum_{j=1}^{N} r_j \leq 1$ holds. We aim to service the packets of different flows at a rate that is proportional to their reservations. As an example shown in Fig. 11, the incoming packets belong to two different flows, with their reservations defined as $(r_1, r_2) = (0.75, 0.25)$. To guarantee that flow 1 is serviced three times faster than flow 2, we need to schedule the packets as follows: in each scheduling round, the total number of bytes that are processed for

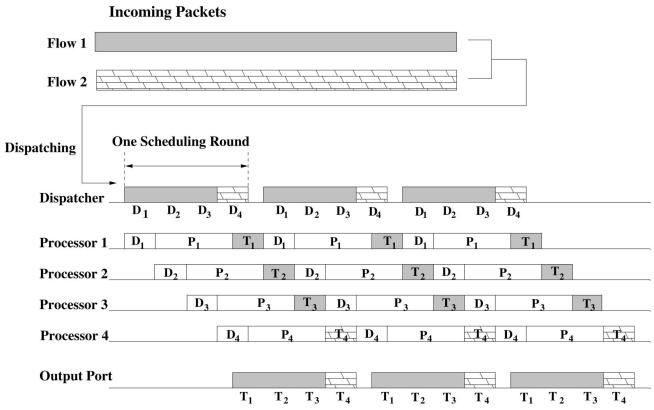


Fig. 11. Fair scheduling among multiple flows.

flow 1 is three times that of the flow 2. Note that the P-ORR algorithm has the property that *the scheduling order of the packets at the dispatching processor is maintained at the transmitting processor*. Therefore, as long as the packets of different flows are scheduled for processing proportionally to their reservations, the packets are delivered orderly and proportionally, as illustrated in Fig. 11.

To extend the P-ORR algorithm to handle multiple flows, the basic idea is given as follows: Given the batch size B and the flow reservations $(r_1, r_2, \ldots, r_N)$, the number of bytes that are scheduled in each round for flow f_j $(j = 1, 2, \ldots, N)$, denoted as F_j, is

$$F_j = r_j \times B. \tag{20}$$

To perform a practical scheduling, we must guarantee that at least one packet is dispatched from any individual flow. Hence, another constraint must be set for the minimal schedulable batch size I as follows:

$$\min_j(r_j \times I) \geq L. \tag{21}$$

Therefore, I should be redefined as $I = CL$, where

$$C = \left\lceil \frac{1}{\min(\min_i(\alpha_i), \min_j(r_j))} \right\rceil. \tag{22}$$

Once I is determined, B is set as a multiple of I. F_j denotes the bytes that should be scheduled from a flow in a scheduling round. But, the packets may not fit into F_j due to their variable lengths. Hence, we adopt the packet management policy described in the above section to deal with variable length packets. Let $FBalance_j$ be the remaining number of bytes that should be dispatched to flow f_j, a packet with the size $PSize$ is dispatched only if $FBalance_j \geq PSize/2$.

Let the load distribution among N flows be $(F_1, F_2, \ldots, F_N)$ and the load distribution among M servers be $(Q_1, Q_2, \ldots, Q_M)$. When the packet scheduler operates, it keeps two pointers: one pointer, i, points to the processor that is currently receiving packets; the other pointer, j, points to the flow that is currently being serviced. The scheduler also keeps two balance counters: one counter, $Balance_i$, records the remaining number of bytes that should be dispatched to the processor p_i; the other counter,

Fair Scheduling of Multiple Flows Using P-ORR

Parameter Initialization:

(1) Using (2) and (3), determine α_i, $\forall i$.
(2) Using (22), determine the minimal batch size I.
(3) Set the batch size B as a multiple of I. Determine the number of bytes that should be scheduled to each worker processor in a scheduling round. $Q_i = \alpha_i B$, $\forall i$.
(4) Using (12), determine the gap Gap_d that should be adopted between the initiation of two adjacent batches.
(5) Determine the number of bytes that should be scheduled for each flow. $F_j = r_j B$, $\forall j$.

Packet Scheduling:

```
i ← 1    /* Let i point to processor p_1 */
j ← 1    /* Let j point to flow f_1 */
Balance_i ← Q_i
FBalance_j ← F_j
while ( true )
    if (all flows are inactive)   /* Non-backlog processing */
        Q_i ← α_i B, ∀i
        i ← 1
        Balance_i ← Q_i
        wait for Gap_d seconds
    else
        while (flow f_j is inactive)
            FBalance_j ← r_j B
            j ← (j + 1)%N
        PSize ← size of the head-of-line packet of flow f_j
        if (Balance_i < PSize/2)
            Q_i ← Q_i + Balance_i
            if (i == M)
                    i ← 1
                    Balance_i ← Q_i
                    wait for Gap_d seconds
            else
                    i ← i + 1
                    Balance_i ← Q_i
        if (FBalance_j < PSize/2)
            F_j ← F_j + FBalance_j
            j ← (j + 1)%N
            FBalance_j ← F_j
        else
            Dispatch the head-of-line packet of flow f_j to p_i
            Balance_i ← Balance_i − PSize
            FBalance_j ← FBalance_j − PSize
```

Fig. 12. Fair scheduling of multiple flows using P-ORR.

$FBalance_j$, records the remaining number of bytes that should be serviced for the flow f_j. When the packet scheduler looks at the head-of-line packet of flow f_j, it compares $PSize$ to both $Balance_i$ and $FBalance_j$. The packet is scheduled only if

$$\min(Balance_i, FBalance_j) \geq PSize/2. \tag{23}$$

The algorithm to handle multiple flows using P-ORR is illustrated in Fig. 12. The algorithm achieves both load balancing among processors and fair scheduling among flows.

6 Simulation Results and Discussion

In this section, we verify the analytical results, given in the earlier sections, through rigorous simulations. First, we study performance of P-ORR and highlight certain intrinsic

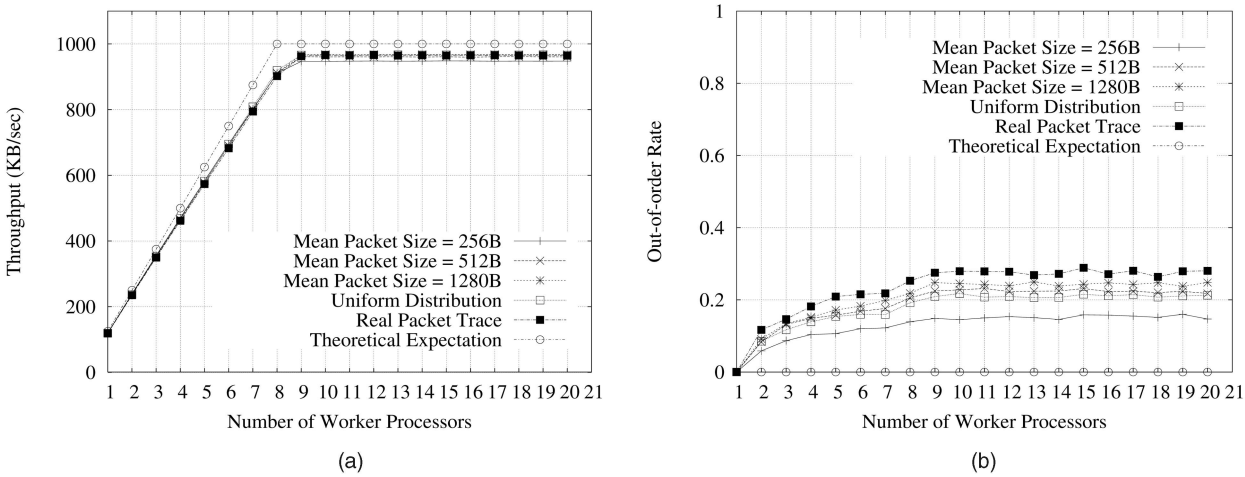


Fig. 13. Experimental verification of P-ORR on homogeneous systems. (a) Throughput verification of P-ORR. (b) Out-of-order rate verification of P-ORR.

advantages of it. Next, we compare P-ORR with several other strategies in the literature to explicitly show the performance gain expected by P-ORR. Last, we demonstrate the effectiveness of P-ORR to handle multiple incoming flows with different reservations.

We developed a software simulator to model a heterogeneous NP system with one dispatching processor, multiple worker processors, and one transmitting processor. The simulator is designed to generate and process packets with variable size, which is confined in between 20 and 1,500 bytes. Three packet size models are used in our simulation study: 1) exponential distribution with three mean packet size, 256 bytes, 512 bytes, and 1,280 bytes; 2) uniform distribution; and 3) real packet trace obtained from [24].

Both homogeneous and heterogeneous backlogged systems are studied here. The homogeneous system is configured to be consistent with previous theoretical analysis, where $z_r = z_s = 1\ \mu\text{s/byte},\ w_i = 6\ \mu\text{s/byte},\ \forall i$. The heterogeneous system is configured with $z_r = z_s = 1\ \mu\text{s/byte}$, $w_i = 4\mu\text{s/byte}$, $\forall i$, $i \bmod 4 = 1$, $w_i = 6\ \mu\text{s/byte}$, $\forall i$, $i \bmod 4 = 2$, $w_i = 8\ \mu\text{s/byte}$, $\forall i$, $i \bmod 4 = 3$, $w_i = 10\ \mu\text{s/byte}$, $\forall i$, $i \bmod 4 = 0$. The number of worker processors M is varied from 1 to 20 to observe the performance.

6.1 Performance of the Packetized ORR Algorithm

To observe the performance of P-ORR for processing variable length packets in a realistic system, we conduct the simulation with all three packet length distribution models mentioned above. The simulation results are compared with the analytical results obtained in Section 3.4. Figs. 13 and 14 show the variation of the throughput and out-of-order rate as a function of the number of worker processors on the abovementioned homogeneous and heterogeneous systems, respectively. Note that the theoretically expected throughput in Fig. 13a is originally illustrated in Fig. 9c in Section 3.4. The throughput of variable packet sizes closely matches the analytical result. But, the saturation throughput achieved is slightly lower than the theoretical expectation. This is because, in our simulation, the scheduler guarantees that there is no overlapped processing on any processor, which may cause

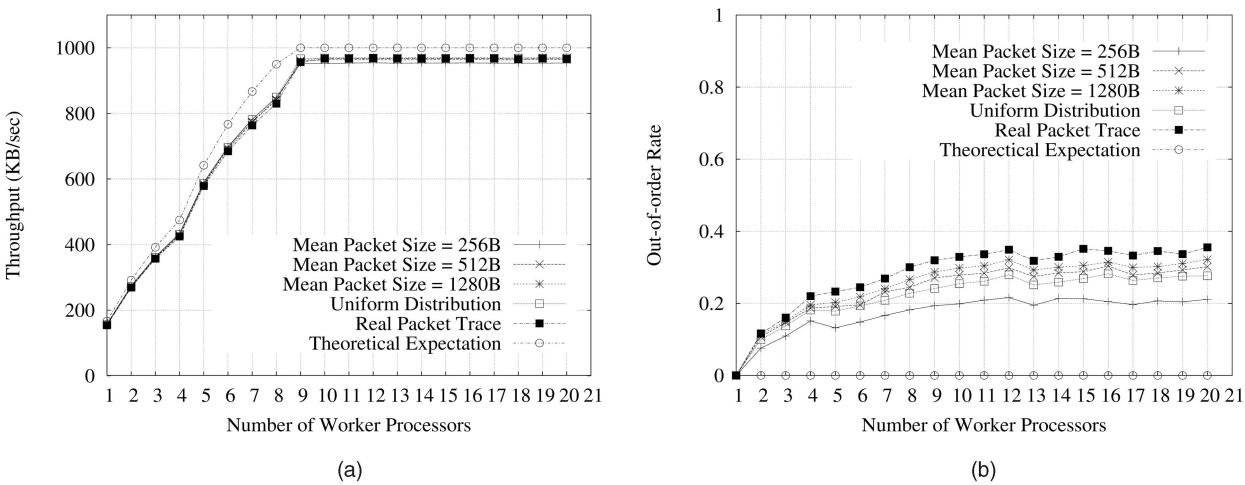


Fig. 14. Experimental verification of P-ORR on heterogeneous systems. (a) Throughput verification of P-ORR. (b) Out-of-order rate verification of P-ORR.

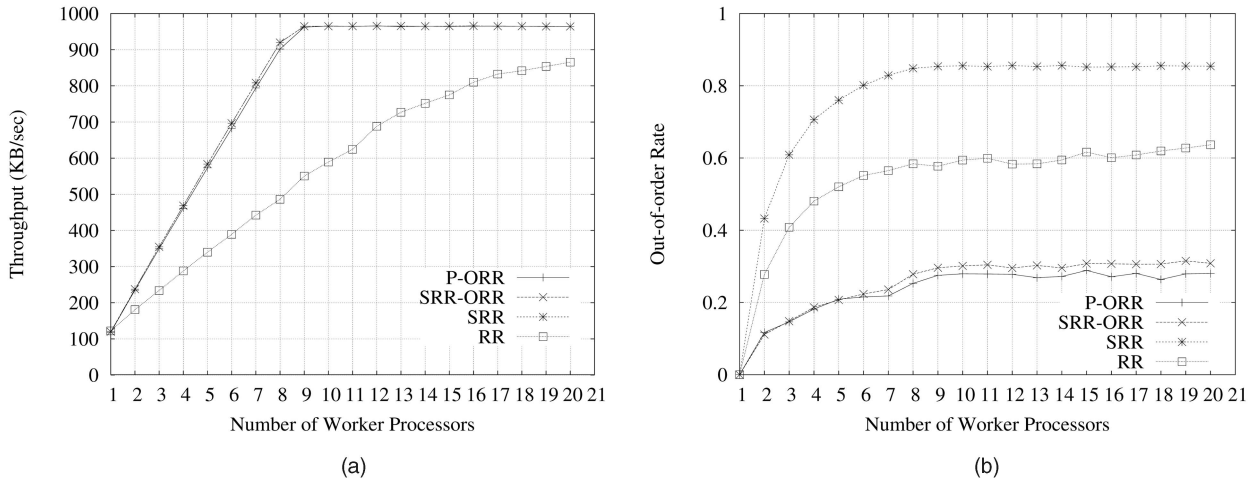


Fig. 15. Experimental comparison among different scheduling schemes. (a) Throughput of different scheduling schemes. (b) Out-of-order rate of different scheduling schemes.

p_d to have a longer gap time than the calculated one under variable packet length situation. The out-of-order rate is higher than theory, which is zero. This discrepancy is reasonable because the theoretical load distribution cannot be strictly guaranteed in the presence of variable length packets. We also note that the out-of-order rate increases with the number of processors as expected.

In all four figures, the curves of different packet size distributions overlap with each other, indicating the adaptation capability of P-ORR. The similarity between the homogeneous and heterogeneous system results shows that P-ORR is capable of dealing with various processor configurations. It is also interesting to note that the saturating point of throughput occurs when there are nine worker processors in the realistic system, while the theoretically proved saturating point is 8. This discrepancy is again caused by variable packet lengths that lead to deviation of the actual load distribution from the ideal load distribution. After the saturating point, when more processors are used, no more throughput gain is observed and the out-of-order rate tends to increase. Therefore, we need to select a proper number of processors that guarantees high throughput and incurs tolerable out-of-order rate.

6.2 Performance Comparison with Other Schemes

Now, let us compare P-ORR with several other load-sharing schemes like SRR [15] and ordinary RR. The homogeneous configuration is used in this study. The ideas of SRR and RR are implemented as specified in the literature. With the RR scheme, one packet is dispatched to each processor in each round without considering the variety of packet size. With the SRR scheme, the total number of bytes dispatched to each processor is proportional to their communication bandwidths. SRR and RR do not consider any communication overhead between the dispatching/transmitting and worker processors in their design. To obtain a fair comparison, we introduce the same communication delay $z_r = z_s = 1\ \mu\text{s/byte}$ for each scheme as modeled in P-ORR. Also, we can design a combined strategy of SRR and ORR called SRR-ORR. In this scheme, we distribute the load among worker processors according to our ORR algorithm but apply SRR to deal with variable length packets. In other words, P-ORR and SRR-ORR are two different packetized versions of ORR.

As shown in Figs. 15a and 15b, P-ORR achieves the highest throughput and the smallest out-of-order rate among all the schemes. RR offers the worst throughput due to the potential load imbalance incurred by blindly dispatching packets. By adapting the load distribution to accommodate variable packet length, P-ORR, SRR-ORR, and SRR achieve comparable throughput because good load balancing is ensured. On the other hand, P-ORR and SRR-ORR greatly reduce the out-of-order rate over SRR and RR because ORR maintains an in-order delivery pattern while balancing the load, whereas SRR and RR do not consider in-order delivery at all. The advantage of P-ORR over SRR-ORR can be observed by the relatively lower out-of-order rate. In conclusion, P-ORR outperforms all other schemes in both load balancing and sequential delivery. While taking extra care to minimize the out-of-order rate, P-ORR still produces the highest throughput.

6.3 Fair Scheduling of Multiple Flows Using P-ORR

Let there be six flows and each flow makes a different reservation, defined as (0.3, 0.3, 0.1, 0.1, 0.1, 0.1). To evaluate the effectiveness of P-ORR algorithm to provide fair scheduling among multiple flows, we generate different flows at the same arrival rate and observe if the service rate of each flow is controlled by its reservation. Again, the study is conducted on the homogeneous system configuration. Fig. 16a shows the total throughput and each flow's individual throughput as a function of the number of processors. Clearly, the total throughput is fairly shared among the flows. All the six flows are serviced at a rate proportional to their reservations even though they arrive at the same rate. Hence, a fair scheduling among flows is successfully achieved by our algorithm. Note that, while scheduling among multiple flows, the system still produces comparable total throughput as that presented in Fig. 13a, indicating good scalability under heavy workload. Fig. 16b demonstrates each flow's out-of-order rate as a function of

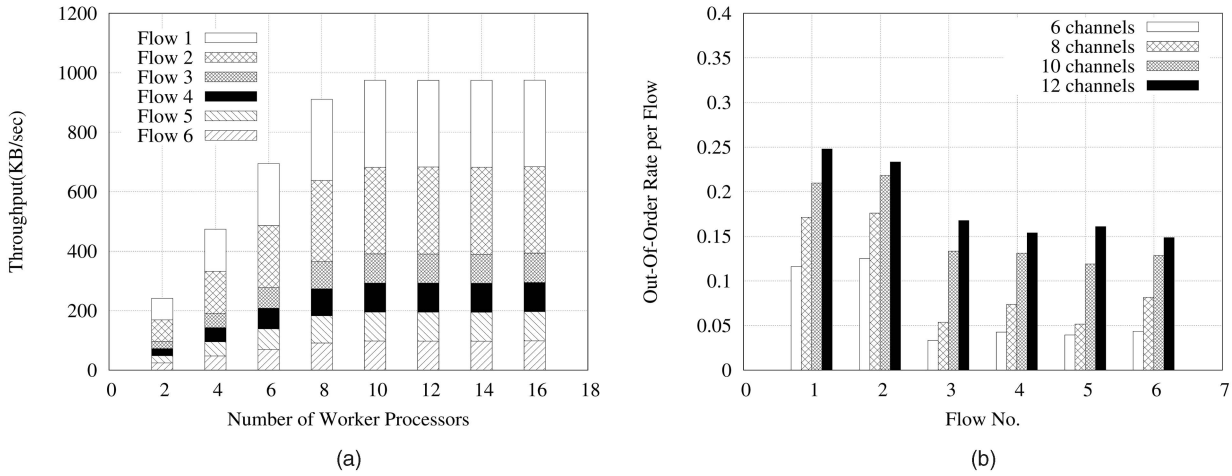


Fig. 16. Scheduling multiple flows. (a) Throughput of multiple flows. (b) Out-of-order rate of multiple flows.

the number of processors. The out-of-order rate per flow slightly increases as the number of processors increases. The flow with higher reservation tends to have higher out-of-order rate because larger portion of its packets are serviced and involved in the out-of-order delivery. Overall, the out-of-order rate falls into a small range, which is similar to the out-of-order rate presented in Fig. 13b.

7 Conclusion

In this paper, we have proposed an efficient packet scheduling algorithm, P-ORR, that is capable of scheduling variable length packets among a group of heterogeneous processors to ensure both load balancing and minimal out-of-order packet delivery. P-ORR is based on the ORR algorithm, which we developed from rigorous theoretical derivation by assuming that the workload is perfectly divisible. Several important theoretical results for ORR were presented in this paper. We also provided extensive sensitivity results through analysis and simulation to show that the proposed algorithms satisfy both the load balancing and in-order requirements for packet transmission. For multiple flows, we extend P-ORR to service packets according to each flow's reservation. Simulation results have verified that fair scheduling among multiple flows is successfully achieved.

Future work includes the design of an optimal mapping between multiple flows and processors and the derivation of fairness bounds. In order to minimize the performance discrepancy between ORR and P-ORR, we are also working on a new packetized technique to better address the problem of variable packet length.

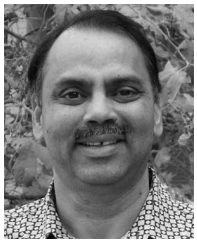
References

[1] G. Welling, M. Ott, and S. Mathur, "A Cluster-Based Active Router Architecture," *IEEE Micro,* vol. 21, no. 1, Jan./Feb. 2001.
[2] Intel, Intel ixp2800 Network Processor, http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm, 2008.
[3] IBM, "The Network Processor: Enabling Technology for High-Performance Networking," 1999.
[4] Motorola, Motorola C-Port Corporation: C-5 Digital Communications Processor, http://www.cportcom.com/solutions/docs/c5brief.pdf, 1999.
[5] E. Blanton and M. Allman, "On Making TCP More Robust to Packet Reordering," *ACM SIGCOMM Computer Comm. Rev.,* vol. 32, pp. 20-30, Jan. 2002.
[6] S. Bohacek, J.P. Hespanha, and J. Lee, "A New TCP for Persistent Packet Reordering," *IEEE/ACM Trans. Networking,* vol. 14, no. 2, pp. 369-382, Apr. 2006.
[7] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, "Building Robust Software Based Router Using Network Processors," *Proc. 18th Symp. Operating Systems Principles (SOSP '01),* pp. 216-229, Nov. 2001.
[8] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *Computer,* vol. 23, no. 5, May 1990.
[9] L. Kencl and J.Y.L. Boudec, "Adaptive Load Sharing for Network Processors," *Proc. IEEE INFOCOM,* 2002.
[10] J. Guo, F. Chen, L. Bhuyan, and R. Kumar, "A Cluster-Based Active Router Architecture Supporting Video/Audio Stream Transcoding Services," *Proc. 17th Int'l Parallel and Distributed Processing Symp. (IPDPS '03),* Apr. 2003.
[11] B.A. Shirazi, A.R. Hurson, and K.M. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE CS Press, 1995.
[12] J. Guo, J. Yao, and L. Bhuyan, "An Efficient Packet Scheduling Algorithm in Network Processors," *Proc. IEEE INFOCOM '05,* vol. 2, pp. 807-818, Mar. 2005.
[13] B. Wu, Y. Xu, H. Lu, and B. Liu, "An Efficient Scheduling Mechanism with Flow-Based Packet Reordering in a High-Speed Network Processor," *Proc. IEEE Workshop High Performance Switching and Routing (HPSR),* 2005.
[14] W. Shi and L. Kencl, "Sequence-Preserving Adaptive Load Balancers," *Proc. ACM/IEEE Symp. Architectures for Networking and Comm. Systems (ANCS '06),* pp. 143-152, Dec. 2006.
[15] H. Adiseshu, G. Parulkar, and G. Varghese, "A Reliable and Scalable Striping Protocol," *Proc. ACM SIGCOMM '96,* pp. 131-141, 1996.
[16] J.A. Cobb and M. Lin, "A Theory of Multi-Channel Schedulers for Quality of Service," *J. High Speed Networks,* vol. 12, nos. 1, 2, pp. 61-86, 2002.
[17] S. Iyer, A. Awadallah, and N. McKeown, "Analysis of a Packet Switch with Memories Running at Slower than the Line Rate," *Proc. IEEE INFOCOM '00,* pp. 529-537, 2000.
[18] D. Khotimsky and S. Krishnan, "Evaluation of Open-Loop Sequence Control Schemes for Multi-Path Switch ES," *Proc. IEEE Int'l Conf. Comm. (ICC '02),* vol. 4, pp. 2116-2120, 2002.
[19] Y.C. Cheng and T.G. Robertazzi, "Distributed Computation with Communication Delays," *IEEE Trans. Aerospace and Electronic Systems,* vol. 24, no. 6, pp. 700-712, Nov. 1988.
[20] V. Bharadwaj, D. Ghose, V. Mani, and T.G. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems.* IEEE CS Press, 1996.
[21] T.G. Robertazzi, "Ten Reasons to Use Divisible Load Theory," *Computer,* vol. 36, no. 5, pp. 63-68, May 2003.

[22] M. Shreedhar and G. Varghese, "Efficient Fair Queuing Using Deficit Round Robin," *IEEE/ACM Trans. Networking*, vol. 4, pp. 375-385, June 1996.
[23] T. Wolf and M. Franklin, "Design Tradeoffs for Embedded Network Processors," *Proc. IEEE Int'l Conf. Architecture of Computing Systems (ARCS '02)*, vol. 2299, pp. 149-164, Apr. 2002.
[24] Cooperative Assoc. for Internet Data Analysis, http://www.caida.org, 2008.

Jingnan Yao received the BS degree in communications engineering from Xidian University, Xi'an, China, the MS degree in electrical and computer engineering from the National University of Singapore, and the PhD degree in computer science from the University of California, Riverside. She is currently with the Wireless and Security Technical Group, Cisco Systems. Her research interests include network processors, load scheduling in parallel and distributed systems, and peer-to-peer computing.

Jiani Guo received the BE and ME degrees in computer science and engineering from Huazhong University of Science and Technology, Wuhan, China, and the PhD degree in computer science from the University of California, Riverside, in 2006. She has been with the Datacenter, Switching and Services Group, Cisco Systems since July 2006. Her research interests include active router, network processors, regular expression acceleration, job scheduling in parallel and distributed systems, and cluster computing. She is a member of the IEEE and the IEEE Computer Society.



Laxmi Narayan Bhuyan has been a professor since January 2001 and is the chairman of the Computer Science and Engineering Department, University of California, Riverside (UCR). Prior to that, he was a professor of computer science at Texas A&M University (1989-2000) and program director of the Computer System Architecture Program at the US National Science Foundation (1998-2000). He has also worked as a consultant to Intel and HP Laboratories. His current research interests are in the areas of computer architecture, network processors, Internet routers, and parallel and distributed processing. He has published more than 150 papers in related areas in reputed journals and conference proceedings. He is currently the editor in chief of the *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. His brief biography and recent publications can be found at his web page at http://www.cs.ucr.edu/~bhuyan/. He is a fellow of the IEEE, the ACM, and the American Association for the Advancement of Science (AAAS) and a member of the IEEE Computer Society.