A Scalable Per-flow Priority Scheduling Scheme for High-Speed Network

Guodong Li[1, 2], Zhen chen[2, 3], Anan Luo[1, 2], Yibo Xue[2, 3], and Chuang Lin[1, 3]
[1]*Dept. Computer Science and Technology, Tsinghua University, Beijing, China*
[2]*Research Institute of Information Technology, Tsinghua University, Beijing, China*
[3]*Tsinghua National Lab for Information Science and Technology, Beijing, China*
guodongli07@gmail.com, laa@mails.tsinghua.edu.cn
{zhenchen, yiboxue, chlin}@tsinghua.edu.cn

***Abstract*—In order to guarantee latency and bandwidth of application, packet scheduling is considered as a crucial module in network device. However, it is hard to maintain and schedule millions of queues for in-progress flows at link speed in high speed network. In this paper, we propose a scalable per-flow scheduling scheme named DQS-SPQ-DRR (Dynamic Queue Sharing-Strict Priority Queue-Deficit Round Robin), which requires only a small size of fast memory to achieve fine-grained service guarantee. The scheduling scheme is in a scalable hierarchical manner, in which the first layer supplies service differentiation and the second guarantees bandwidth and latency. A limited number of queues are dynamically shared among concurrent flows based on the fact that the number of simultaneous flows is only in hundreds no matter what the link speed is. Experiments on DQS-SPQ-DRR carried out based on real and synthetic traces demonstrate it well fit in small memory space and ensure per-flow service.**

Keywords: scheduling, active flow, QoS

I. INTRODUCTION

Packet scheduling is a key technique to guarantee the networking service for critical applications. To design an effective and efficient scheduling scheme, several issues should be addressed: 1) how to allocate bandwidth to each flow on demand; 2) how to guarantee the quality of service for critical applications; 3) how to deploy the scheduling schemes easily.

Using a dedicated queue for each flow, a constant-time scheduler such as Deficit Round-Robin (DRR) [1] can provide good service guarantees for applications. Unfortunately, the number of in-progress flows can be extremely large. For example, experiments on an OC48 link trace show that there are about 0.9 million 5-tuples flows in a 5-minute interval. On 40-Gbps links, there can easily excess a million in a shorter observation interval [4]. Caching states for millions of flows is a big challenge for high speed networking devices. If the states are stored in SRAM, the amount of SRAM required for worst-case is often both infeasible and impractical; if they are kept in DRAM, the state lookup and update are too slow. Therefore, how to effectively organize the queues in small size of fast memory and schedule them in different priorities efficiently is a significant but unsettled issue.

Many researchers [2-4] focus on queuing or counting flows at a time scale of seconds or minutes. However, a packet is buffered in high speed device only for several microseconds mostly, so scheduling active flows (have packets in their queues at a microsecond time scale) may be a good idea. Recent discovery shows that the number of concurrent active flows is only in hundreds at microsecond scale [5]. Based on this observation, it is possible to use only hundreds of queues and share them among active flows [6], rather than keep track of all in-progress flows with millions of entries in a full state table. Therefore a data structure called active flow list (AFL) [5] can be employed to store active flows. When the first packet of a flow arrives, an empty queue is allocated and a new entry is inserted to AFL; when the queue becomes empty, the relative entry in AFL is deleted and the queue is freed and can be reassigned to another newly incoming flow.

In this paper, a novel scalable per-flow scheduling scheme is proposed. AFL is used to store active flows in a small size of fast memory. The scheduler is hierarchically organized for enqueue and dequeue operation. The first layer provides service differentiation by distributing flows into different priority groups. The second provides latency and bandwidth guarantee for each flow which prevents one flow from occupying too much resource. It is a scalable scheme and each layer can use existing scheduling algorithms to achieve service differentiation and guarantee. Our results show that all memory required by this scheme is small enough to be well held in fast memory; quality of critical service is improved compared to original solution.

The rest of the paper is organized as follows. Section II introduces the related work. Section III presents the scalable hierarchical scheduling scheme. Section IV gives a named DQS-SPQ-DRR implementation. Section V discusses the experimental results. Section VI concludes the paper and future works.

II. RELATED WORK

Packet scheduling has been studied extensively and many scheduling algorithms and architectures are given.

IntServ [7] is the pioneer of scheduling architecture. It reserves resource for all in-progress flows. It can achieve well per-flow service guarantees. However, it has to maintain all state information for all flows in its route. Its sophisticated implementation is not feasible to the huge number of flows. So it is not widely used in Internet.

A. Nikologiannis et al. [8] and A. Ioannou et al. [9] introduced special hardware to implement thousands of queues for per-flow queuing to provide advanced service guarantee respectively. As it requires application specific integrated circuit (ASIC) for queue organization and scheduling, these methods take high hardware cost and long developing cycle, therefore cannot scale up with the swift development of the network.

S. Floyd et al. [10] presented Class Based Queuing (CBQ), which had been implemented in many Linux distributions [11]. It introduced hierarchical link sharing to allow multiple agencies, protocol families, or traffic types to share the bandwidth on a link in a controllable fashion. Link sharing was organized in tree structure, where each node represented one share such as agencies or policy. On the basis of CBQ, HPFQ (Hierarchical Packet Fair Queuing) [12] and HFSC (Hierarchical Fair Service Curve) [13] were introduced to achieve fair queuing. However, these methods only dealt with the flow aggregation. They would have to deepen the tree in order to get finer grained service guarantee, with significant increase in scheduling cost.

A. Kortebi et al. [5] firstly discovered the fact that the number of in-progress flows was in millions and increased with the link speed, but the number of active flows was only in hundreds. However, [5] did not take the opportunity to design a new scheduling scheme, and only presented a simple fair queue schedule scheme which cannot provide sufficient service guarantee.

III. THE DESIGN OF PACKET SCHEDULING SCHEME

A. Scheduling design

The objectives of our packet scheduling scheme are: 1) to provide per-flow queuing in SRAM; 2) to provide latency and bandwidth guarantee for critical flows; 3) to be adaptable to existing scheduling algorithms; 4) to be easy to implement. To achieve these, we combine AFL, priority group, and bandwidth guarantee algorithm together.

In order to providing per-flow queuing, AFL is employed. Queues are allocated for active flows only. When a packet whose associated flow priority is obtained from preconfigured policy arrives, a lookup action in AFL is triggered. If the lookup returns unsuccessful result, it means that the flow is not already recorded in AFL and thus can be treated as a new flow. The queue manager allocates a new empty queue from the free queue stack for the flow. Then the flow identifier (such as the 5-tuples or their hash value) and its scheduling information is composed as a new entry and inserted into AFL. If successful, queue manager updates the scheduling information of the matching entry in AFL. Finally, the packet is inserted into its corresponding queue. With this method, active flows are dynamically mapped to finite physical queues, so only a small number of physical queues are needed.

Our scheduling scheme is organized in hierarchical manner, in order to provide service differentiation. Priority group is the first layer. Usually, higher priority group would get more chance to send packets. Various existing packet scheduling algorithms that provide service differentiation can be employed here. For example, to prevent low priority groups from starving, weighted DRR (WDRR) scheduling can be implemented to offer services among different groups proportional to their assigned weight.

The second layer is in priority group, and the candidate scheduling algorithms should treat flows fairly and prevent a small number flows from occupying too much resources. DRR is a suitable for this use, because it is easy to implement by hardware or software and an O (1) fair scheduling scheme for latency and bandwidth guarantee.

The structure of the scheme is depicted in Figure 1. Suppose there are N priority groups and 6 active flows currently in this example. Flow 1, 2, 4 are served in priority group 0, flow 3 in group k and flow9, 11 in group N. Only the packet number is shown here as the scheduling information (short for sch. info. in Figure 1). Queues dynamically insert to priority group and are organized as linked-list for *DRR* scheduling. The packets in a queue are organized as linked-list too. In fact, only the header and rear packet pointer is stored in SRAM.

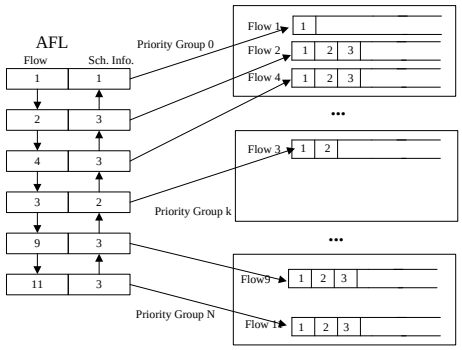


Figure 1. Scheduling structure.

In summary, our proposed scheduling scheme leverages on AFL to maintain a small number of mappings between active flows and physical queues, priority group to differentiate services, and DRR to provide latency and bandwidth guarantee. Other existing popular scheduling algorithms can also substitute DRR flexibly to achieve required results depending on various needs of applications.

B. Scheduling algorithms

To simplify the description, we adopt the strict priority group, i.e., the low priority group is not scheduled until all the packets in high priority ones have been sent out. In each group, we employ DRR algorithms. The scheduling information in AFL includes number of in queue packets $PktsNum_i$, current credit DC_i, quantum Qua_i. Priority group should maintain the AFL entry pointer f_i to get DRR scheduling information, current packet number $PktsNum_i$, and $lastDequeueRound_i$. Figure 2 and Figure 3 present the pseudo-codes of enqueue and dequeue algorithms.

Enqueue Operation: On the arrival of a packet p, it firstly gets the priority in order to find priority group (line 3). If packet p does not belong to any active flow, it allocate Q_k from the free queue management stack and set $PksNum_i$ to 1 and initial the DRR quantum and DC_i, then the new entry inserts to the tail of AFL(lines 5-10). Otherwise it simply adds the $PktsNum_i$ (lines 12). After searching AFL, insert packet p to relative queue Q_k in priority group *pri* (lines 14-15). If it is the first packet in this group, to set the bit in bitmap Priority to 1(lines 16-17).

```
Enqueue Algorithm
1   On arrival of packet p;
2   i = ExtractFlow(p);
3   pri = GetPriorityFlow(p);
4   Search f_i->q_k in AFL;
5   If null
6         allocate a new queue Q_k;
7         i->PktsNum = 1;
8         i->Quantum = Qua_i
9         i->DC = 0;
10        InsertAFL(f_i,Q_k);
11  Else
12        i->PktsNum++;
13  End If
14  Insert(p,Q_k);
15  pri->PktsNum++;
16  If(pri->packet==1)
17        Priority[pri]=1;
18  End If
```

Figure 2. Enqueue Module.

Dequeue Operation: It firstly gets the highest priority group *K* that is not empty (line2), and then extracts the packet *p* at the head of *lastDeqRound* queue (line 3). Adding a Qua_i to the DC_i of the flow, if DC_i is larger than the *p's* length, to send *p* out and decrease DC_i by the packet size (line 6-7). The loop allows the flow to emit up to DC_i bytes. If the packet in this flow becomes empty, AFL deletes the entry and breaks the loop (lines 5-14). After the loop, l*astDequeRound* in group *K* updates (line 15). At last if the packet in group *K* becomes zero, the *K-th* bit in bitmap Priority is set to 0 (16-18).

```
Dequeue Algorithm
1   While(1)
2         Find the first not empty priority group K,;
3         p=Headof(K->lastDeqRound,&fi);
4         fi->DC += fi->Quantum;
5         While(fi->DC > p->length)
6               Dequeue(K->lastDequeueRound);
7               fi->DC -= pac->length;
8               K->PksNum--;
9               fi->pkts--;
10              If(fi->pkts ==0)
11                    Delete the fi entry in AFL;
12                    break;
13              End If
14        End While
15        K->lastDequeuRound++;
16        if(K->PksNum ==0)
17              Priority[K]=0;
18        End if
19  End While
```

Figure 3. Dequeue Module.

IV. SCHEME IMPLEMENTATION

We have implemented the above scheduling scheme by integrating Dynamic Queue Sharing (DQS) [6], Strict Priority Queue (SPQ), and enqueue-time DRR [16], which is called DQS-SPQ-DRR. Please note that, the implemented scheduler works in a per-flow manner, and all the control information can fit in SRAM.

A. DQS

To speed up the searching and updating on the active flow list, hash is utilized to divide the whole AFL into multi sub-list. In this case, flows with the same hash value are directed to the same sub-list, so the operation is executed in small range. More details of DQS can be found in [6].

B. SPQ

SPQ is the easiest scheme to achieve service differentiation. Each flow gets its priority from preconfigured policy, so when it arrives in AFL, the assigned queue for the flow is inserted into the corresponding priority group to which it belongs.

Since SPQ is from high to low priority, dequeuue scheduling may check for packets in empty group which degrades the performance. To solve the problem, one bitmap indicating which group currently has packets is maintained. Instead of polling all groups, it simply read the bitmap and search for bits that are set. Especially many modern processors families (such as Intel's IXP, Cavium's OCTEON) support the instruction FFS (Find First Set) [14, 15]. I searches one bitmap for the location of the first bit that is set. With the solution, the group selection processing only searches a bit vector to determine the current state efficiently.

C. Enqueue-time DRR

The state-of-the-art network processor is usually multi-core architecture, so it is good at processing packets in parallel. For example, there can be several processes or threads to send packets to different queues. DRR dequeue processing, however, does not have the same characteristics. Each round of DRR needs to run sequentially. It does not allow for one queue to start the next round until all other queues have finished the current round. Hence dequeue processing has to be always in a single thread or process.

Under this condition, with more than one core performing enqueue and only one core doing dequeue, the unbalance results in the need to simplify dequeue process. We introduce the enqueue-time DRR which sorts the packets into DRR rounds at enqueue-time. So it moves the load of schedule handling from deque-time to enqueue-time. More details about enqueue-time DRR can be found in [16].

D. Complexity analysis

The memory required for DQS-SPQ-DRR is quite small. Let's denote each entry as flowing:

DQS: it only stores the mapping of queue and active flows. Here, the number of hash slots is *W*; the number of physical of queue is *N*; the number of active flows is *L*; the number of bits to identify a flow is *B*; the number of bits for *DC* and Quantum is *Q*, for *PktsNum* and *BytesNum* are both *C*. Then the memory requirement for the mapping scheme when sub-tables are organized in double linked-list.

$$M_{DQS} = W + L\,(2\log L + \log N + B + 2Q + 2C) \quad (1)$$

SPQ: It only needs maintain the priority bitmap and some scheduling information. The layer of priority group is *P*; the number of bits for *PktsNum* is *C* too; the max *DequeRound* is *D*; the number of bits for *CurEntry* is *I*.

$$M_{PQ} = P + P(C+\log D+I) \quad (2)$$

Enqueue-time DRR: For each round, it should maintain the head and tail of pointer for each dequeue round. As denoted above, the memory required is:

$$M_{DRR} = 2PDI \quad (3)$$

So the total memory need is

$$M = M_{DQS} + M_{PQ} + M_{DRR} \quad (4)$$

For example given *M=512* slots, *N=512* queues, active flow *L= 512, B=32, Q =32, C=32.P=64, D = 64, I=32.* The memory required is only about 363Kbit and can be implemented in SRAM easily.

The time cost of this scheme includes enqueue time and dequeue time. Search, insert and delete AFL accounts a big portion of enqueue time. According to [6], we can get:

$$T_{search} = 1 + (L-1)/2M \quad (5)$$

$$T_{insert} = L/M \quad (6)$$

$$T_{delete} = 1 \quad (7)$$

The enqueue time cost of priority queue and enqueue-time DRR are both O (1). The total enqueue time cost is this, i.e.

$$T_{enqueue} = MAX\ (T_{search}, T_{insert}) + T_{PQ} + T_{DRR} \quad (8)$$

Dequeue procedure is easy. The scheduler goes through per-round linked-list sequentially and sends out packets from each round linked list. If the queue becomes empty, the relative entry in AFL is deleted. The time cost is O (1), i.e.

$$T_{dequeue} = T_{delete} + T_{PQ} + T_{DRR} \quad (9)$$

Here we also set *M=512* slots and *L= 512* flows as mentioned above. The enqueue time is 2.5 and dequeue time is 1.

V. PERFORMANCE EVALUATION

We evaluate the performance of DQS-SPQ-DRR by means of synthetic traffic and Internet trace data.

A. Original trace statistics

We use three real traces from [17]:

a) NLANR1: It was collected on November 24st, 2002 at the University of Leipzig Internet OC3 access link.

b) NLANR2: It was collected on August 14st, 2002 at the OC48 link from IPLS Abilene router towards CLEV.

c) NLANR3: It was collected on June 1st, 2004 at the OC192 link from IPLS Abilene router node towards KSCY.

Here we set the statistic time scale to be 10 milliseconds. Output bandwidth is shared among all the active flow and the output capacities are set to make the traffic load of each trace to be 0.95, as shown in Table 1. The load is defined as the ratio of the average rate and the output bandwidth. From the table, we get that average rate of each trace is far less than its original output capacity.

TABLE I. PACKET TRACE STATISTICS SUMMARY.

Trace	**NLANR1**	**NLANR2**	**NLANR3**
average trace rate	14.4Mbps	372Mbps	557Mbps
original output capacity	155Mbps	2.5Gbps	10Gbps
regulate output capacity	15.2Mbps	392Mbps	586Mbps
packets number	10M	20M	100M
flows in progress	53K	75K	100K
MTU	1500byte	1537bytes	9000bytes

The complementary distribution of AFL size is shown in Figure 4, which indicates the number of active flows. We observe that it is only in the number of hundred (256 in the worst case), much less than the number of flows in progress.

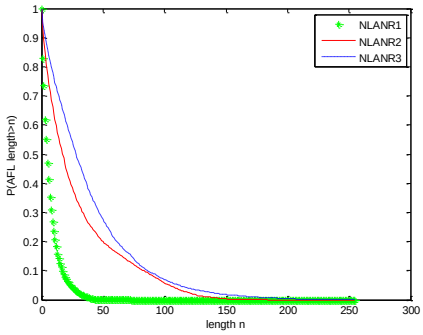
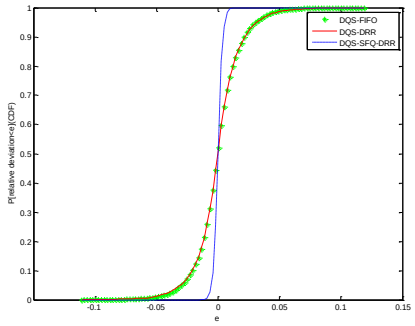
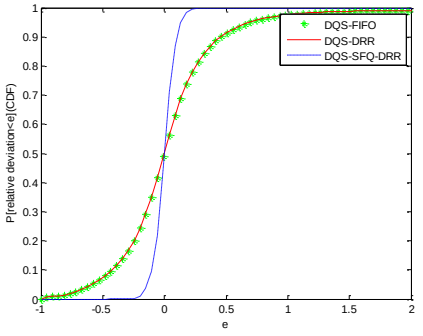


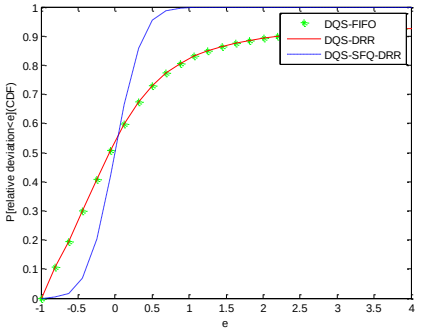
Figure 4. Complementary distribution of AFL size.



(a) Relative deviation of 1Mbps flow



(b) Relative deviation of 32Mbps flow



(c) Relative deviation of 128Mbps flow

Figure 5. CDF of relative deviation for flows of various rate

B. Performance results

The following experiment results about the three trace data are similar. In order to simplify the description, NLANR3 with the highest speed is selected to show the performance. We add six constant rate flows to the NLANR3. The rates of the flows are 1Mbps, 8, 16, 32, 64 and 128 Mbps and the packet size is 1024 bytes. As a result, to get traffic load of 0.95, the link capacity is regulated to 848Mbps. To guarantee latency and bandwidth of these 6 flows, trace flows are set lower priority than them.

To verify the performance of DQS-SPQ-DRR, we compare it with DQS-FIFO that each packet is scheduled as its enqueue sequence and DQS-DRR that packets are scheduled in round robin manner without service differentiation.

In the following, we focus on bandwidth and latency of these six flows. The expected latency of a flow is defined as the average incoming interval among packets, i.e.

Expected = packet length/flow rate (10)

To compare the result of these three algorithms, we define "relative deviation" to show the deviation between experiment and expected result.

Relative deviation= (experiment – expected)/expected (11)

Here we only present the comparisons of 1Mbps, 32Mbps and 128Mbps flows. The same thing happens for the other three flows.

Figure 5 shows the cumulative distribution of relative deviation. We can get that DQS-SPQ-DRR performs the best among the three schemes no mater of the protected flow rate. DQS-FIFO and DQS-DRR perform almost the same, because both of them don't treat flows different and flows are not protected finely. Another fact is that smaller rate flows can get better service guarantee for all the algorithms. For example, about 90% of the interval is exact to expected interval with 1Mbps flow in DQS-SFQ-DRR. If the flow rate goes to 128Mbps, the relative deviation is more dramatic. However, more than 95% of the interval falls in the rage [-0.5, 0.5] for DQS-SFQ-DRR.

VI. CONCLUSION

We propose a novel scalable per-flow scheduling scheme which use a small fast memory to achieve fine-grained service guarantee. The queue and scheduler's data structure can be all stored in SRAM. It is a per-flow queuing scheme by only maintaining dynamic queue for active flows. The scheduler is organized as a hierarchical manner, in which the first layer providing service differentiations and the second does the service guarantee.

The advantages of this architecture lies in 1) it only maintains a small number queues to achieve per-flow queuing; 2) it is a scalable hierarchical architecture and compatible to existing scheduler algorithms; 3) it can use SRAM to achieve the best performance for high speed network.

An instance implementation called DQS-SPQ-DRR is presented to evaluate the performance. Trace-driven experiment shows that under DQS-SPQ-DRR, the AFL length is still in the number of hundreds. The guaranteed flow acquires its service quarto no matter of the variation of the other background traffic.

ACKNOWLEDGMENT

This work has been supported by the National High-Tech R&D Program (863 Program) of China under grant No.2007AA01Z468. The authors would like to thanks Yaxuan Qi, Baohua Yang and other colleagues in my lab for their help. We also thank Chengchen Hu for his valuable suggestion

REFERENCES

[1] M. Shreedhar, and G. Varghese, "Efficient Fair Queuing Using Deficit Round-Robin," IEEE/ACM Trans. on Networking, vol. 4, pp. 375-385, 1996.

[2] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. Diot, "Packet-level Traffic Measurements From The Sprint IP Backbone," IEEE Network, vol. 17 pp. 6-16, 2003.

[3] N. Hua, B. Lin, J. Xu, and H. Zhao, "BRICK: A Novel Exact Active Statistics Counter Architecture," Proc. ANCS 2008.

[4] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement," Proc. SIGMETRICS, 2008.

[5] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts, "Evaluating the Number of Active Flows in a Scheduler Realizing Fair Statistical Bandwidth Sharing," Proc. SIGMETRICS 2005.

[6] C. Hu, Y. Tang, X. Chen, and B. Liu, "Per-flow Queuing by Dynamic Queue Sharing," Proc. INFOCOM 2007.

[7] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: an Overview," RFC1633, 1994.

[8] A. Nikologiannis, and M. Katevenis, "Efficient Per-flow Queueing in DRAM at OC-192 Line Rate Using Out-of-order Execution Techniques. Proc. ICC 2001.

[9] A. Ioannou, and M.Katevenis, "Pipelined Heap (Priority Queue) Management for Advanced Scheduleing in High-speed Networks," IEEE/ACM Trans. on Networking, vol. 15, pp. 450-461, 2007.

[10] S.Floyd, and V. Jacobson, "Link-sharing and Resource Management Models for Packet Networks," IEEE/ACM Trans. on Networking, 1995.

[11] Main page of CBQ, www.icir.org/floyd/cbq.html

[12] J. Bennett, and H Zhang, "Hierarchical Packet Fair Queueing Algorithms," IEEE/ACM Trans. on Networking, vol. 5, pp. 675-689 , 1997.

[13] I. Stoica, H. Zhang, T. S. E. Ng, "A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service," IEEE/ACM Trans. on Networking, vol. 8, pp. 185-199, 2000.

[14] Official website of Cavium about OCTEON product family, www.cavium.com/OCTEON_MIPS64.html

[15] E. J. Johnson, A. R. Kunze, "IXP2400/2800 Programming," INTEL PRESS, 2004.

[16] U. R. Naik, and P. R. Chandra, "Designing High-Performance Networking Applications," INTEL PRESS, 2004.

[17] NLANR, "Passive measurement and analysis (PMA)," [Online]. Available: http://pma.nlanr.net.