Design and Evaluation of Diffserv Functionalities in the MPLS Edge Router Architecture

Wei-Chu Lai, Kuo-Ching Wu, and Ting-Chao Hou*

Center for Telecommunication Research and Department of Electrical Engineering

National Chung Cheng University

Chia-Yi, Taiwan, 62102

* *tch@ee.ccu.edu.tw*

Abstract—Differentiated Service (DiffServ) in combination with Multi-Protocol Label Switching (MPLS) is a promising technology in converting the best-effort Internet into a QoS-capable network. This paper describes the design and implementation of key DiffServ components, including classifier, meter/shaper, queue manager, and scheduler, in an MPLS edge router architecture on a network-processor platform. We describe how the DiffServ functionalities were realized and also analyze the performance of the system under different traffic patterns. Various factors that cause performance degradation in the architecture are observed and analyzed. These include receiving processing architecture, transmitting processing architecture, micro-engine processing power, memory access latency, and complexity of each DiffServ component. Among them, we found that traffic classification demands more processing resource and hence is a major factor in limiting the system throughput.

Keywords: QoS, DiffServ, Network Processor, MPLS

1. Introduction

The current Internet is based on the connectionless IP technology which provides best effort services. IP Routers treat all packets equally and hence cannot give preferential treatment to high priority traffic streams. Routers in the IP network also routes packets based on some shortest distance or lowest cost algorithm in forwarding packets to their destinations. The practice of selecting a shortest path without regard to current link utilization usually makes certain links or paths congested with traffic even though there are other paths with slightly longer distance and under-utilized capacity. Lack of Quality of Service (QoS) support in the current Internet hinders the introduction of multimedia and QoS sensitive services to the users.

DiffServ [1] (Differentiated Service) as proposed in IETF provides a solution for supporting QoS in IP networks. Instead of providing different services based on per-microflow state as in the Integrated Service (IntServ), DiffServ treats different flows which reside in the same class in the same manner and provides a more scalable solution for QoS in IP networks by restricting the complex functions such as packet classifier and traffic conditioner within edge routers.

To overcome the link congestion and resource usage inefficiency problem caused by shortest path IP routing, the adoption of Traffic Engineering [2] is needed. Traffic Engineering enhances the performance and utilization of an operational network at both the traffic and resource levels. Service provider networks with Traffic Engineering capability could accommodate much more customer traffic without upgrading network infrastructures and provide a more reliable service to their customers. However, Traffic Engineering is difficult to achieve in networks with the IP protocol due to its connectionless feature. Hence, for providing Traffic Engineering capability effectively, MPLS [3] is introduced by the Internet Engineering Task Force (IETF).

MPLS is an important technology for the next generation networks. The idea of MPLS is to bring the connection-oriented concept into a connectionless IP network. DiffServ in combination with MPLS is a promising network architecture the enables the transport of novel QoS sensitive applications and services. This paper describes how we design the four key components (classifier, traffic conditioner, queue manager, and scheduler) of DiffServ in an MPLS router architecture which is built on the Network Processor platform. We also analyze the performance of the system under different traffic patterns and present our findings on where the performance bottlenecks are under different traffic scenarios.

The rest of the paper is organized as follows. In section 2, we describe the basic operation of DiffServ and MPLS. In Section 3, we give a brief overview of the network processor (Intel IXP1200) used in this work. In section 4, the design of the four Diffserv components is explained. Performance evaluation of the completed DiffServ MPLS router architecture is provided in section 5. Section 6 is the summary of this work.

2. DiffServ and MPLS

2.1. Diffserv

When a packet enters a DiffServ domain and is received at an edge router, the packet is classified firstly to find which Service Level Agreement (SLA) the packet belongs to. Next, it is checked whether its sending rate is greater than the one committed by the service provider or not through the traffic access control procedure. The packet is dropped or shaped if any over-sending conditions occur. Then, the packet is marked with a special tag referred to as DSCP (DiffServ Code Point) in the IP header to determine the Per-Hop-Behavior (PHB) if the packet is not dropped. Finally, the router stores and sends the packet by using suitable queue management and scheduling mechanisms to provide service differentiation among different traffic classes indicated by PHB.

After the marked packet is sent from the edge device, the core router within the DiffServ domain could identify the PHB for the packet from the DSCP tag and avoid the more complex process of packet classification and traffic conditioning.

2.2. MPLS

For supporting the MPLS protocol, a new header referred to as shim header [4] is added between layer 2 MAC header and layer 3 IP header in a packet. Once an IP packet is assigned to a particular Forwarding Equivalence Class (FEC) corresponding to a routing table entry or a classification rule, the packet is inserted with a shim header in which a specific index known as a "label" is encoded for the FEC before being forwarded. After labeling and transmitting the packet at the first router (a.k.a. the ingress router), the subsequent routers just process the packet with a simple lookup at the label switching table in which the next hop and a new label are recorded for the FEC, thus avoiding the relative complex routing table lookup procedure. The packet is stored and forwarded to the next-hop router after the old label tagged by the previous-hop router is replaced with new one recorded in the label switching table. The procedure of label lookup and label swapping continues at each hop until the last router removes the shim header from the received packet, looks up the IP header and makes a decision for transmitting the packet to a host or a router in adjacent domain. The route on which a labeled packet travels is referred to as a Label Switched Path (LSP) and is decided by means of label lookup results at each router. The content of the label switching table at each router is set up by the signaling protocol.

3. Network Processor: Intel IXP1200

The Network Processor is a new technology which provides an alternative to traditional software or hardware solutions in network equipment design [5]. It has flexible, low developing cost, and high-performance features. What Network Processor can do includes traditional Internet equipment such as routers, switches and firewalls, and newer devices such as Voice over IP (VoIP) gateways, VPN edge devices, wireless Access Points (APs), etc.

The IXP1200 is the first generation of Intel Network Processor and provides wire-speed processing for OC-3 to OC-12 multi-service network applications. The architecture of the IXP1200, shown in Fig. 1, consists of one 32-bit Intel StrongARM Core, six multi-threading, parallel-packet processing Microengines, access interfaces for SRAM and SDRAM memory, a PCI bus, and an IX Bus for transferring data from/to MAC device [6]. The StrongARM core is a full 32-bit RISC processor core that can be used for management functions, running routing protocols, exception handling, and other tasks. At boot period, StrongARM is responsible for loading the system kernel from Flash ROM or via networks, to initialize peripherals and for loading the microcode into Microengines.

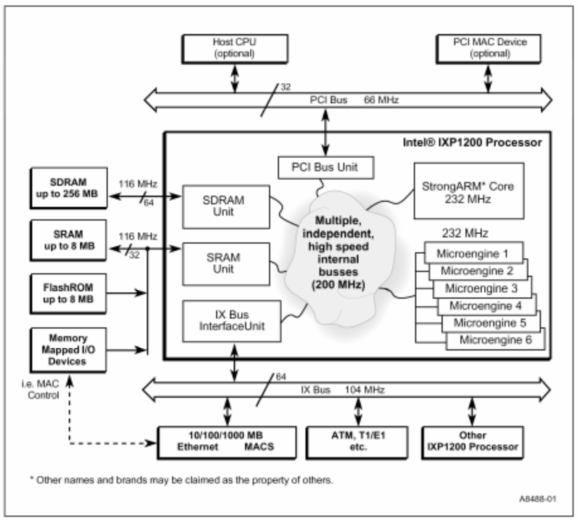

Fig. 1 Intel IXP 1200 architecture.

Six Microengines are fully programmable 32-bit engines with a 5-stage execution pipeline and are used for any function requiring high-speed packet inspection, data manipulation, and data transfer.

Once an incoming packet received at the MAC device is detected by the IX Bus Unit, the IX Bus Unit notifies the thread in the Microengine, which is responsible for processing the packet. Then, the thread commands the IX Bus Unit to move the packet from the MAC device to the Receive FIFO residing at IX Bus Unit and copy the packet header into Microengine registers for routing table lookup or any other further processing task. After processing the packet header, the thread commands SDRAM Unit to move the entire packet from Receive FIFO to SDRAM, waiting to be transmitted. At the same time, some threads responsible for transmitting packets continue to check whether there is any packet waiting for forwarding or not. Once a transmitting thread detects that a packet is stored in SDRAM and waits to be forwarded, the thread moves the packet into the Transmitting FIFO corresponding to the output port and commands the IX Bus to transfer the packet to a specific MAC device.

4. Design and Implementation

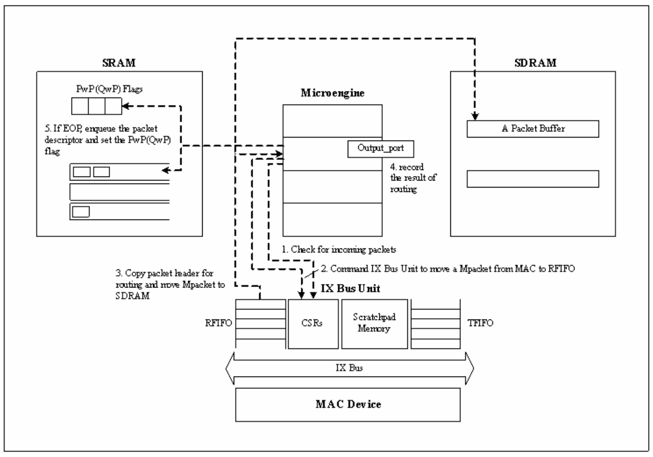
SRAM
PwP(QwP) Flags
5. If EOP, enqueue the packet descriptor and set the PwP(QwP) flag
Microengine
Output_port
4. record the result of routing
SDRAM
A Packet Buffer
1. Check for incoming packets
2. Command IX Bus Unit to move a Mpacket from MAC to RFIFO
3. Copy packet header for routing and move Mpacket to SDRAM
IX Bus Unit
RFIFO
CSRs
Scratchpad Memory
TFIFO
IX Bus
MAC Device

Fig. 2 Packet receiving flow.

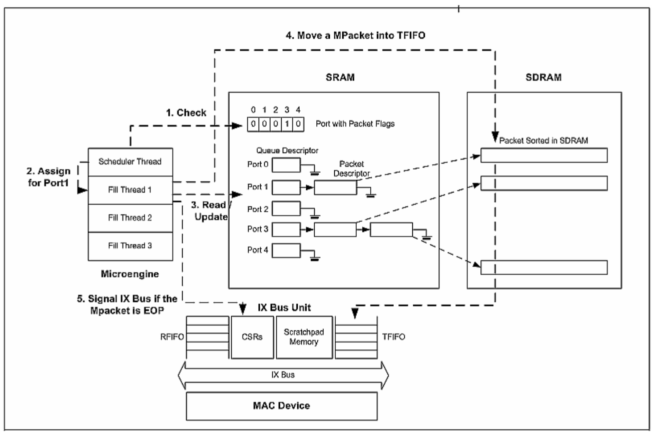

Fig. 3 Packet transmitting flow.

4.1. Packet receiving flow

In our DiffServ MPLS router design, we allocate two Mircoengines with a total of eight threads to be responsible for receiving packets from eight 10/100M Ethernet ports. Each of the eight threads is dedicated to one Ethernet port. The thread continues to check the receiving ready flag in the IX Bus Unit, corresponding to the port for which it is responsible. If a ready flag is set, meaning that there is an incoming packet received from the MAC device, the thread first commands the IX Bus Unit to move an Mpacket[1] from the MAC device to the Receiving FIFO (RFIFO), and next commands the SDRAM Unit to move the Mpacket from RFIFO to an allocated SDRAM buffer. If the Mpacket is a Start-of-Packet (SOP) Mpacket, the thread commands the IX Bus to send a copy of the packet header to the registers in the Microengine for routing table lookup. After determining which outgoing port the packet should be sent to, the thread records the information in an Output_Port register and continues to receive the remaining Mpackets, if any. If the thread receives an Mpacket which is an End-of-Packet (EOP) Mpacket, it creates a packet descriptor which records the packet size and the allocated buffer in the SDRAM and enqueues the packet descriptor to a proper outgoing queue in the SRAM according to the information in the Output_Port register.

4.2. Packet transmitting flow

Fig. 3 illustrates the key components of the packet transmitting flow on IXP1200. A microengine consisting of four threads which execute alternately is allocated to eight 10/100M Ethernet ports for transmitting packets. Among the four threads in the Microengine that is assigned for transmitting, one thread is assigned as the scheduler thread which is responsible for checking a set of special flags in a specific SRAM address to determine whether there is any nonempty queue waiting for processing. A queue is dedicated to an outgoing port. If a queue is nonempty, the corresponding flag will be set, and the scheduler thread will assign one of the other three threads, referred to as fill threads, to transmit the packet in the nonempty queue after detecting the flag is set. After receiving the assignment for a specific queue, a fill thread reads the queue descriptor for the specific queue in which the linked-list head/tail pointers and the number of packets in the queue are recorded. The fill thread fetches the head-of-queue packet descriptor (the first element of the linked list), in which the fill thread can get information about the address of the packet content in the SDRAM, the total number of Mpackets and the valid bytes of the last Mpacket, and removes the packet descriptor from the linked list. Once the fill thread gets the

[1] The IXP1200 processes the packet in a fixed length, 64 bytes, unit. A packet with a length greater than 64 bytes is fragmented into many pieces referred to as Mpackets.

above information it signals the SDRAM Unit to move the first Mpacket of the entire packet to the Transmit FIFO (TFIFO), sets the control information such as the outgoing port, validate length, etc., in the TFIFO and signals the IX Bus Unit to move the Mpacket to the MAC device once SDRAM Unit completes moving the entire packet. If the packet length is greater than one Mpacket, the fill thread records the SDRAM address of the packet content, the number of remaining Mpackets and the length of the last Mpacket in a register for later use by other threads in continuing processing the packet.

4.3. Scheduler: Deficit Round Robin

Deficit Round Robin (DRR) [7] is the scheduling algorithm we adopted for the Diffserv scheduler. For supporting the DRR, we create eight queues for each port and the corresponding “Queue with Packet” (QwP) flags. The modification means the corresponding flag will be set if a queue, rather than a port, has packets to be forwarded. Furthermore, we allocate an area in the SRAM serving as a table for recording the quantum size for each queue. The Deficit Count (DC) is placed in the queue descriptor for a queue.

At boot time, a management routine running at the StrongARM core initializes the quantum size for each queue. After initialization, the scheduler thread continues to check the QwP flags. If one or more flags that belong to the same port are set, the scheduler thread assigns work to one of the three fill threads and passes the port information (which port is waiting for service) and eight QwP flags for the outgoing port (which queue in the port has packets). Once the fill thread receives an assignment it reads a Port_Queue register in the Microengine to check which queue within the port is served last time. The packet in the queue served last time within the port should have priority to be forwarded based on the DRR algorithm. After determining which queue to serve, the fill thread reads the corresponding queue descriptor for which the format is shown in Fig. 4. The packet count field is the total number of packets waiting in the queue. The head and tail pointers are the addresses of the first and the last packet descriptors in the SRAM, respectively. The DC field is the Deficit Count of the DRR for the queue. The Average Queue Size field records the average number of packets waiting in the queue for the queue management to be discussed later.

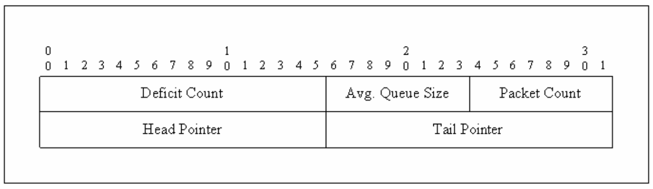

Fig. 4 The format of a queue descriptor.

Through using the head pointer in the queue descriptor, the fill thread fetches the head-of-queue packet descriptor to calculate the packet length. If the corresponding DC of the queue is greater than the length of the packet, the fill thread processes the packet as described in 4.2. If the DC is smaller than the packet length, the fill thread serves the next queue within the port. Which next queue has packets to be served can be inferred from QwP flags.

4.4. Queue Manager: Weighted RED

Weighted Random Early Detection (WRED) [8] is the adopted queue manager for DiffServ. The first implementation issue is how we drop a packet according to a specified probability. The solution we adopt is using a pseudo random vector (PRV) in which a set of bits is set randomly for a given probability. For example, to use a PRV with a 32-bit width to represent a given probability, 0.25, the 32 bits of PRV may be set as “0100 0110 0000 0010” in which a quarter of 32 bits are set and the others are cleared. Furthermore, we use a rotational mask with only one bit set to perform the “AND” operation with a PRV to find which bit in a PRV is currently valid. For example, if the mask is currently “0000 0001 0000 0000”, and a packet arrives, then the result of the AND operation is “0000 0000 0000 0000”, and the packet does not need to be dropped. Each time an AND operation is performed the mask circularly shift left for one bit to use the next bit in a PRV next time.

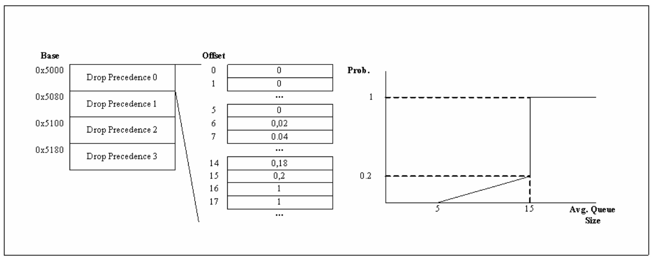

Fig. 5 Using PRVs to represent a RED probability curve.

Moreover, we use a set of PRVs to approximate a RED probability curve by arranging the PRVs with different probabilities within an area of SRAM as depicted in Fig. 5. As a packet arrives, the router chooses a PRV according the current average queue size and uses a rotational mask to determine whether to drop the packet or not.

4.5. Traffic Conditioner: Token Bucket

The Token Bucket traffic access control mechanism is used as the DiffServ traffic conditioner. In our design, we allocate a segment of the SRAM memory to set up a table for token bucket operation, in which each entry consists of a current amount of tokens in the bucket, the maximum bucket size, and the token number added per specified time interval (0.1 sec). The entries from #0 to #511 are dedicated to classified flows when the router is an ingress device. The other entries are dedicated to MPLS labeled flows when the router is a core device. A managing process running at StrongARM core is responsible for updating the amount of tokens in the bucket for each entry in the Token Bucket table periodically. It gets the token increment information from an entry and adds it into current token number in the bucket. If the current token in the bucket exceeds the maximum bucket size, then it is modified to be equal to the maximum bucket size.

4.6. Classifier: Multidimensional Ranging Matching

Multidimensional Range Matching [9] is a packet classification algorithm which is used to parse the packet headers and to classify packets based on administrative policies. The idea of Multidimensional Range Matching is illustrated in Fig. 6. We assume a set of three packet filtering rules in two dimensions (the IP source and destination addresses) are given. For example, the filtering rule #2 is for packets with the IP source address within the range from 140.112.0.0 to 140.112.255.255 and the IP destination address within the range from 140.123.107.0 to 140.123.107.255. Each filtering rule is represented by a rectangle in our example. The rectangles representing different filtering rules are projected to each dimension to construct the non-overlapping intervals determined by the boundaries of these rectangles. There are a maximum of 2*n*+1 non-overlapping intervals created at each axis (dimension) if there are *n* filtering rules. Each interval in a dimension is associated with a vector referred to as Bit Vector (BV), of which the width is equal to the number of filtering rules, and the corresponding bit for a filtering rule in the BV for an interval is set if the range of the filtering rule covers the interval. To find which filtering rules the packet belongs to, the algorithm first finds the intervals at which the packet is located in each dimension. This can be done by using binary search or other efficient search algorithms. After determining the intervals the packet belongs to at each axis, the algorithm extracts the corresponding BVs and uses bit-wise AND upon BVs to find out which rules the packet conforms to.

For resolving the ambiguous situation that a packet conforms to more than one rule, the bits corresponding to each rule in a BV are arranged based on the priorities of these rules. The algorithm finds the highest priority rule by locating the highest priority bit set in the result of the AND operation for the BVs.

The classifier in our design parses five fields in the packet, which are source address, destination address, protocol type in the IP header, and source port and destination port in the TCP/UDP header.

We use an AVL binary tree search [10] to find which interval the packet is located in each dimension. The structure of an element in a binary tree is defined as in Fig. 7. The first field is used to record the boundary value of a filtering rule at an axis. The Left/Right Pointer fields are used as a pointer to child elements on the left/right sides. The Left/Right BV fields are used as a memory offset for a corresponding BV and are valid when the Left/Right Pointer field is null. For example, as illustrated in 0, two filtering rules are set and the binary tree is constructed. If a packet with a specified address, 140.123.107.25, arrives at the router, the classifier begins to look up the binary tree using the address. It is found that 140.123.107.25 is greater than 140.123.107.20, and the classifier looks up the child element in the right side through Right Pointer in the root element. Next, it is found that 140.123.107.25 is smaller than 140.123.107.255, and the classifier tries to look up the child element in the left side of the current element. However, the Left Pointer is null, so the classifier takes the value in the Left BV, which is 3, to get the address of the corresponding BV.

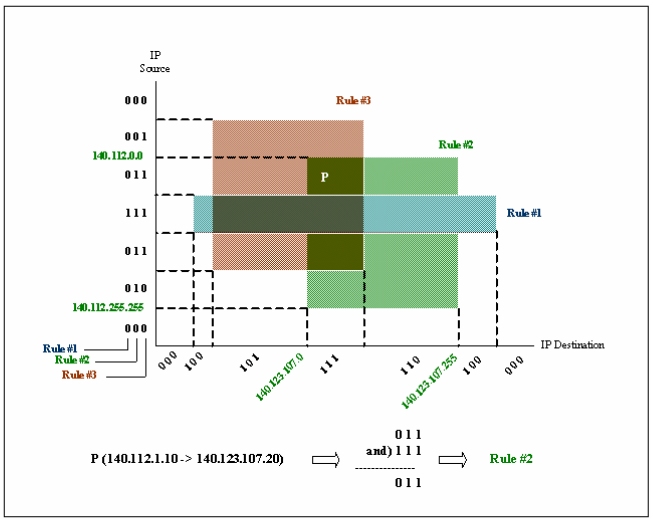

Fig. 6 An example of Multidimensional Range Matching.

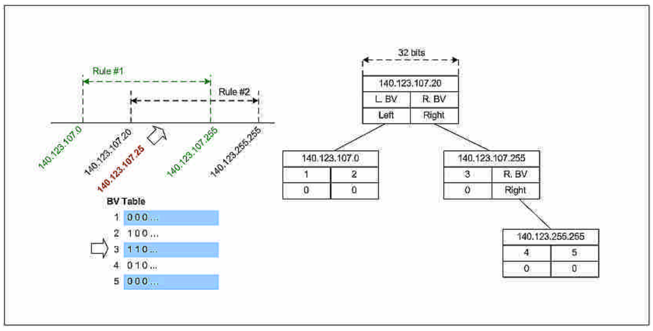

Fig. 7 Using binary tree search to find which interval the packet is located.

5. Performance Evaluation

This section describes how we test our implementation of the DiffServ over MPLS on IXP1200. As depicted in Fig. 8, a SmartBits 6000B is used as a traffic generator. It is responsible for generating specified MPLS or IP traffic patterns to and receiving the returning packets from the IXP1200 to calculate performance result such as throughput and packet loss ratio. A computer is used as a controller to control the SmartBits and to download the linux kernel and the compiled microcode into the IXP1200. Also, a packet analyzing software runs at the computer to snoop the traffic originating from the IXP1200 to get more detailed packet information.

Figure 9 illustrates the throughput of two MPLS flows competing resources at one output port. The DRR scheduling algorithm in our design can allocate different bandwidth to distinct queues in which the two packet flows belong according to the ratio of the quantum for each queue. In our test, the quanta for the two flows in Fig. 9 are set to 3000 and 2000, respectively. The incoming traffic of the two flows is both 60 Mbps and the packet sizes of the two flows are set to 128 bytes. The uppermost line in the graph represents the aggregate throughput at the outgoing port. However, it does not reach the wire speed since the short packet size causes too much MAC layer overhead[2]. The result shows that the scheduler that we implemented on the IXP1200 could achieve specified differentiated bandwidth allocation according to the configuration.

To examine the performance of the RED, a program running at StrongARM core is used to capture the queue size information per micro second. A CBR flow is sent from the SmartBits. The length of packets generated from SmartBits is fixed at 512 bytes and the offered load is slightly greater than the output rate of the IXP1200 to allow accumulation of packets in queue.

Fig. 10 presents the actual queue size and calculated average queue size. The x-axis is time in *u*sec and the y-axis shows the number of packets. The results show that the line representing the actual queue size increases significantly in the beginning. The increasing rate of another line representing the average queue size is slower than that for the actual queue size. The actual queue size decreases when average queue size is over 40 since the RED queue manager starts to drop the packet. However, the actual queue size does not degrade seriously due to the CBR traffic characteristics. The result shows that the RED queue manager we design can be used to restrict the number of packets buffered as well as to accommodate some traffic bursts.

[2] A minimum gap with 960 nsec between Ethernet frames is indicated.

Figure 11 shows a case in which the performance of different forwarding mechanisms such as IP routing, MPLS forwarding, and MPLS fowarding+Classification are tested. Wired speed incoming flows with different packet sizes are generated and sent into a particular 10/100M Ethernet port on the IXP1200. The outgoing flow is transmitted by the IXP1200 from another port after the forwarding decision based on the chosen forwarding mechanism is made. The number of rules for classification is 512. The result shows that the throughputs of different forwarding mechanisms are almost the same and all can not reach the ideal throughput (which is the same as the input rate) even with the longest packet size. A reasonable doubt that the performance bottleneck is not relevant to the forwarding mechanism in the receiving part of a packet processing flow emerges. So, the second test we consider is focusing on the performance of the transmitting part of a packet processing flow. We use a technique in which the transmitting threads on the Microengines skip dequeuing a packet after serving it to make the throughput of transmitting part become independent of the receiving part. Once a packet is enqueued to a specific queue the transmitting threads continue to transmit it, and the queue is viewed as full all the time.

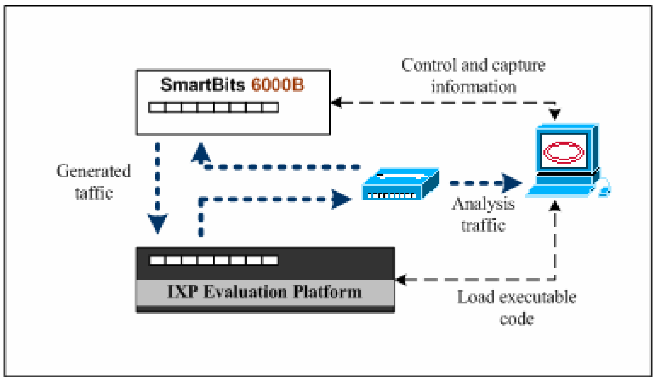

Fig. 8 Test environment.

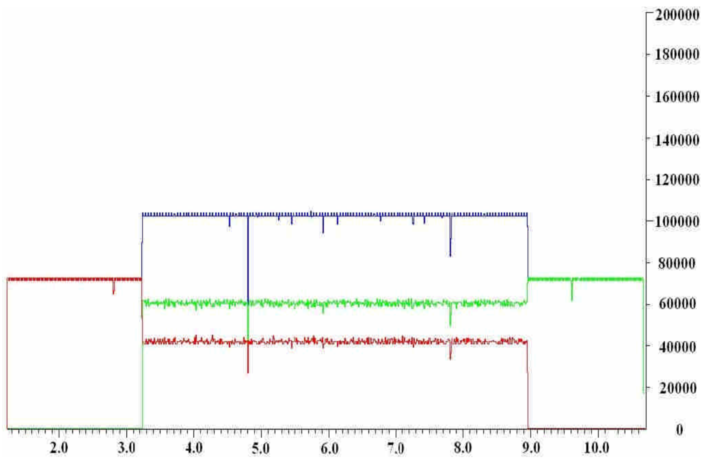

Fig. 9 DRR bandwidth allocation test.

Figure 12 shows the result of transmitting performance test with different packet sizes. The lines of WS1, WS2 … and WS8 are theoretic system throughputs corresponding to one, two … and eight fully-loaded 100M Ethernet ports, respectively. The lines of OUT1, OUT2 … and OUT8 represent the actual measured throughput corresponding to WS1, WS2 … and WS8. A phenomenon that deserves to be mentioned is that the measured system throughputs with six, seven and eight fully loaded 100M Ethernet ports degrade significantly when throughputs approach around 600 Mbps. We found that outgoing packets begin to show frame errors due to wrong layer-2 FCS which is calculated by the hardware on IXP1200. The reason is that the throughputs of the above conditions start to reach the performance upper bound of IXP1200 (OC-12, 622 Mbps) which INTEL claims.

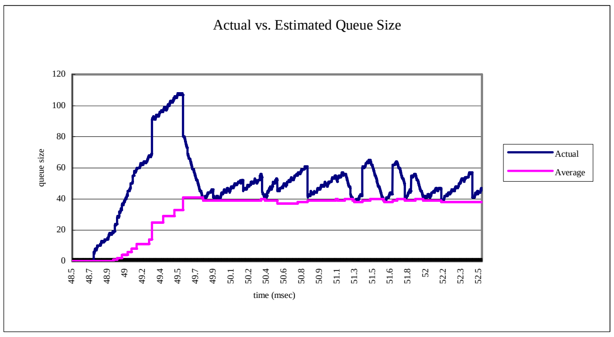

Fig. 10 Queue estimation by RED.

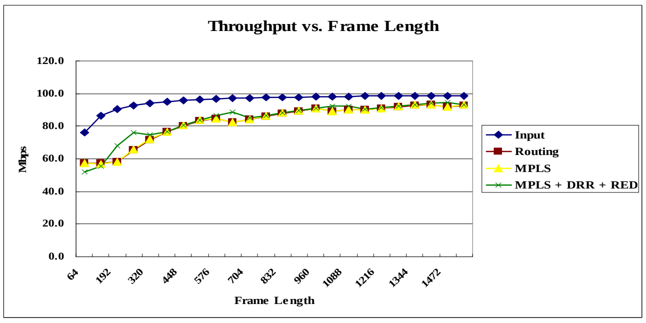

Fig. 11 Throughput vs. frame length (one input, one output).

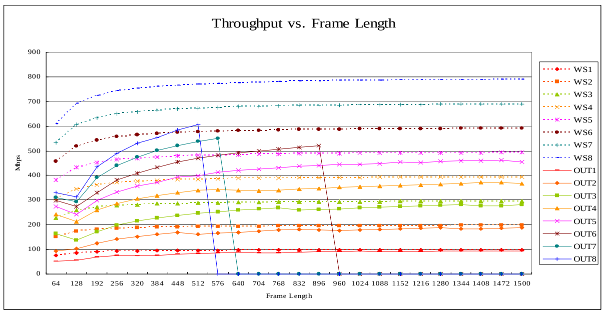

Fig. 12 Performance of the transmitting part on IXP1200.

The result shown in Fig. 14 indicates that the capacity for the transmitting part of packet processing flows is not enough for serving wire-speed traffic, and the result is consistent with our conjecture that the performance bottleneck in Fig. 13 is not the forwarding mechanisms performed at the receiving Microengines.

Another interesting IXP1200 characteristic we found is that even though the IXP1200 can deliver around 500 Mbps total throughput it can not reach the theoretical wire-speed throughput when transmitting data to only one outgoing port. This is due to a system design choice, which requires two lock steps before a packet can be sent out.

For testing the receiving performance without the influence of the transmitting bottleneck, we design a wired-speed aggregated flow consisting of eight micro-flows which enter at one input port and target at one of the eight outgoing ports individually. Figure 13 shows the measured aggregated throughputs on eight ports for different forwarding mechanisms. The number of rules for classification is 512. It is found that all forwarding mechanisms expect the relatively complex forwarding with classification reach ideal throughput for all valid Ethernet packet sizes from 64 bytes to 1500 bytes. After adding classification operations in the forwarding mechanism, the throughput can reach wired speed when the packet size is greater than 196 bytes.

Furthermore, to obtain further information about the performance penalty caused by classification we continue to increase the offered load upon the IXP1200. Figure 14 shows incoming flows at 100, 200, 300 and 400 Mbps. All flows are processed by a Microengine and the corresponding output throughputs are plotted against the frame size. We observe that the total throughput scales when the loading offered by the incoming flow is not higher than 200 Mbps and is close to the upper bound if the packet size is greater than 192 bytes. However, when the total offered load is increased to 300 and 400 Mbps, the increase in throughput is less, i.e. it does not scale. If we distribute the loading over two Microengines, the result becomes significantly different. We find that the throughputs hold a proper ratio relative to their offered load and are close to wire speed when the frame size is greater than 192 bytes. Obviously, the two results show that with classification the computation power of Microengines becomes a bottleneck when the offered load exceeds 200 Mbps.

Finally, as depicted in Fig. 15, we compare the performance of our implementation with the one implemented in [11]. One flow with wire speed and smallest packet size (64 bytes) is input to a 10/100M Ethernet port and destined for another output port where the output data rate is measured. Our implementation achieves better performance at a large number of classification rules (due to possibly a better pipeline design) while both implementations get similar performance at a small number of classification rules.

6. Summary

In this paper, we describe our design and implementation of a DiffServ MPLS edge router with packet classifier, traffic conditioner, queue manager and scheduler on a Network Processor platform, INTEL IXP1200. We tested the performance of our implementation and found that MPLS forwarding does not significantly increase the throughput when compared to IP routing. However, traffic classification does demand more processing resource and hence is a major factor in limiting the system throughput. We also found out that there are more than one throughput bottlenecks in the system and they emerge under different circumstances. In the cases that some output ports are unused, a system design choice causes the transmitting part of the packets forwarding procedure to become a performance bottleneck. Another potential bottleneck is the computation power of the Microengine which limits the throughput at heavy classification loading. The improvement can be made by using more Microengines for classifying packets or raising the Microengines computing capability directly.

Acknowledgement

This work is supported by NSC under the grant NSC-93-2219-E-194-005.

7. Reference

[1] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, "An Architecture for Differentiated Services," IETF RFC2475, December 1998.
[2] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, J. McManus, "Requirements for Traffic Engineering Over MPLS," IETF RFC2702, September 1999.
[3] E. Rosen, A. Viswanathan, R. Callon, "Multiprotocol Label Switching Architecture," IETF RFC3031, January 2001.
[4] E. Rosen, D. Tappan, G. Fedorkow, Y. Rekter, D. Farinacci, T. Li, A. Conta, "MPLS Label Stack Encoding," IETF RFC3032, January 2001.
[5] http://www.netrino.com/Articles/NetworkProcessors/
[6] Intel IXP1200 Network Processor Family – Hardware Reference Manual, Part Number:278303-007, June 2001.
[7] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round-robin," IEEE/ACM Trans. Netw., vol. 4, no. 3, June 1996.
[8] "Quality of Service (QoS) Networking," White paper, Cisco Systems, June 1999.
[9] X. Yang, "Designing Traffic Profiles for bursty Internet Traffic," IEEE Global Telecommunication Conference 2002, November 2002.
[10] G. M. Adel'son-Vel'skii and E. M. Landis. "An algorithm for the organization of information," Solviet Math. Doklady 3:1259-1263, 1962.
[11] Y-D. Lin, Y-N. Lin, S-C. Yang, Y-S. Lin, "DiffServ edge routers over network processors: Implementation and evaluation," IEEE Network, July 2003.

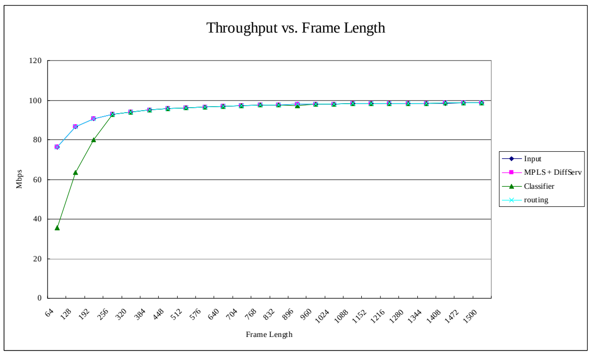

Fig. 13 Throughput vs. frame length (one input, eight outputs).

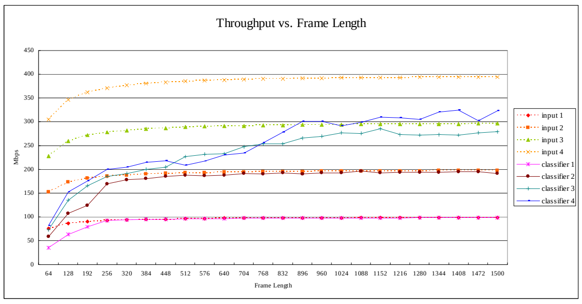

Fig. 14 Throughput vs. frame length (four inputs on one Microengine, eight outputs).

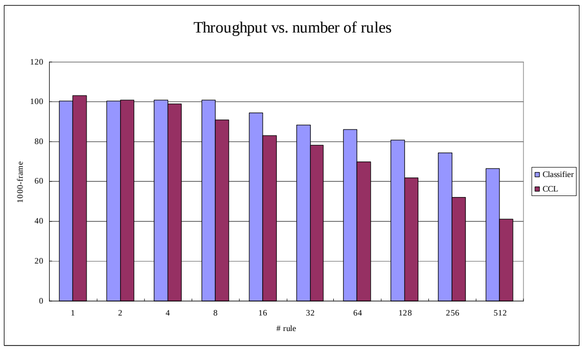

Fig. 15 Performance test with different classifier implementations.