Implementation of Scalable Ethernet-bridge with auto-discovery and auto-updates using network processor

H. P. Shringarpure, G.P. Saraph, and D. Mujumdar

Indian Institute of Technology Bombay, Powai, Mumbai-400076, India

Abstract— **Ethernet is the most widely used LAN technology. The performance of a single large LAN can be enhanced by dividing it into different segments using Ethernet bridging. We implement a scalable Ethernet bridge over IXP1200 network processor. We demonstrate the data plane forwarding functionality of the bridge. The auto-discovery functionality for addition of new nodes and the update functionality have also been implemented. We have exploited the multiprocessing and multithreading capabilities of IXP1200 to obtain an optimized implementation which gives a high throughput at every port.**

Index Terms **— Ethernet bridging, IXP 1200, network processor, auto-discovery.**

I. Introduction

ETHERNET is the most widely used local-area networking technology today. It has proven to be a flexible, durable and scalable technology. Ethernet has seen a ten-fold rise in bandwidth every few years since 1993 as Ethernet (10Mbps) to, Fast Ethernet in 1995 (100Mbps), Gigabit Ethernet in 1997 and finally 10-Gigabit Ethernet in 2002 [1].

Ethernet is cheaper than other options available for setting up a local area network (LAN). It is easier to setup, requires no configuration and is robust to noise. Ethernet uses Carrier Sense Multiple Access with Collision Detect (CSMA/CD) for sharing access for multiple users over the common transmission medium. It is efficient in utilizing the available bandwidth among multiple users in a fair manner. Ethernet has proved to be very efficient at low loads but as the load increases, the performance level of Ethernet decreases [2]. This is because, as the number of stations on a LAN increase, the chances of collision increase. Under the CSMA/CD scheme, when collisions increase, the stations go into contention mode more often. This increases the latency in transmission of data and reduces throughput. Ethernet also has a limitation on the maximum allowable distance for the LAN network based on the round trip propagation time.

To overcome these problems, the network can be divided into multiple segments, using an Ethernet bridge. The bridge forms a star type network by connecting different LAN segments on each of its ports. The bridge passes only those packets destined to segments other than the ones on which they originated. Thus, if the destination and source nodes are on the same segment, the traffic is restricted to that segment only. This helps reduce collisions and improves efficiency of the network. The bridge keeps track of which devices are connected to which port by maintaining a MAC-addresses-to-port mapping table.

Consider the conversion of a single LAN network into a bridged network with four segments.

FIG 1(a). Single LAN network of 4n nodes

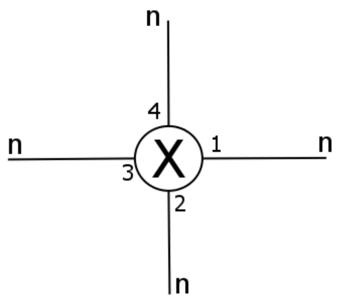

FIG 1(b). Bridged LAN network

For the original LAN topology, consider P_f as the partial usage of the available bandwidth by each node. For 4n nodes the total usage of the network is 4n P_f . The total number of collisions will be proportional to

$$4n \times (4n-1) \times P_f^2 ,$$

where cubic and higher terms are neglected. After dividing the topology into 4 segments of n nodes each, the bandwidth usage of each segment will be n P_f . We assume here that the traffic generated in each segment gets equally distributed to all the segments. Hence one particular segment gets $nP_f/4$ traffic from every other segment. The traffic from other three segments crossing over the bridge to one segment is $3\,nP_f/4$. Thus the total traffic on one segment can be given as

$$nP_f + \frac{3nP_f}{4} = \frac{7nP_f}{4} .$$

So the number of collisions that can take place on a single segment is proportional to P_2 . This analysis can be further extended if we divide the 4n nodes into 4 groups in a logical manner such as departments or subgroups. In that case, we consider the probability of data generated for a source-destination pair within a segment, P_1, to be higher than that for a pair across two segments, P_2. With reference to the topology given in fig[1.b], and considering above probabilities, the total usage on each segment is:

$$P_u = nP_f\left(\frac{P_1}{P_1+3P_2}\right)+nP_f\left(\frac{3P_2}{P_1+3P_2}\right)$$

The probability of collision on the segment is proportional to P_u^2 . Now, if P_2/P_1 is substantially lower than 1, then the overall performance of the network will improve.

The above concept can be further stressed by taking example of an organization like IIT with four departments namely, mechanical, electrical, civil and metallurgy. If all these departments are in a single LAN, then all the intra-department traffic would travel to all departments. Instead, if this network is divided into different departments using a bridge, with n nodes in each department, the intra-department traffic could be very well restricted to the department. At the same time, data transfer could be initiated in other departments simultaneously. The inter-department traffic could be smoothly passed over by the bridge to the required department. The ratio P_2/P_1 will be lower than 1, as departments are logical partitions of the LAN and the crossover of data required, will be less than the intra-department data transfer.

The IXP1200 network processor from Intel provides a very good platform for implementing a network system, as it is optimized for packet processing. It introduces great flexibility in programming the implementation because of its multiprocessing environment. The processor internally has six processors known as microengines, each capable of running four threads. The internal architecture of IXP1200 facilitates reception, processing and transmission of data at fast line rates.

The implementation of an Ethernet bridge over a network processor is dealt with in [ref]. However it is based on a statically built forwarding table. This implementation is not scalable as it does not provide any scheme for adding or deleting any nodes. When such changes take place, this implementation requires updating the MAC address table statically for not just the bridge but for every node in the network. We introduce a fully scalable solution that supports auto-discovery and auto-updates. Our solution also supports variable data rates on any input port of the bridge and uses the hardware resources optimally to give maximum throughput. Making use of the various resources provided by IXP1200, we implement a highly scalable version of the Ethernet Bridge.

II. IXP 1200 NETWORK PROCESSOR

IXP1200 is an integrated Network Processor, consisting of a StrongARM Core Processor, six programmable *Microengines,* standard memory interfaces and high speed bus interfaces. IXP1200 is designed to provide high level of programming flexibility for data packet processing applications. Each *Microengine* has 4 hardware threads, which are effectively used for multiprocessing and for efficiently hiding the memory access latencies. Following is simplified block diagram of IXP1200, depicting its various functional units.

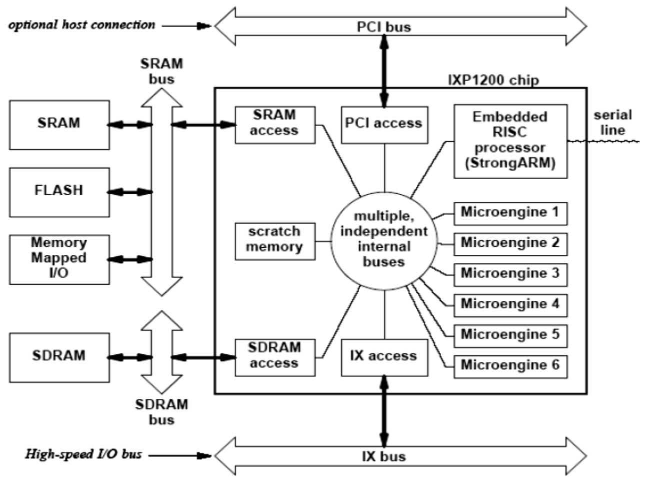

FIG 2 : Architecture of IXP1200 Network Processor

The StrongARM core is a full 32 bit RISC based Processor with integrated caches. It is used for implementing the control plane functionality and high level packet processing. It communicates with the host system through the PCI interface. The host system is used to boot up the StrongARM to run Real-Time Operating System (RTOS) and load programs to be run on microengines. There are six 32-bit, RISC based processors or microengines, optimized for performing packet processing tasks in the IXP. Each *microengine* has four hardware context-switched threads with four independent program counters. Each microengine has fast internal scratchpad memory and high-speed internal bus to the external, shared SRAM and SDRAM memory units. The memory units and microengines are connected to the IX Bus Interface unit that provides access to the external MAC ports through a high speed IX Bus. The interface unit also provides buffers for transmit and receive operations.

The micro-code for the each microengine is developed using IXP1200 Developer Workbench and assembled using assembler before linking. Developer Workbench closely depicts the IXP1200 hardware and provides a suitable platform for debugging and simulation. The simulation environment provides access to all memory locations, thread statistics, data rates and packet counts etc. Various data streams can be simulated using the workbench and thorough testing of the microcode can be done.

III. ETHERNET BRIDGING IMPLEMENTATION

An experimental LAN network with eight nodes is considered. This network is divided into four segments using the Ethernet Bridge. Port0 to port3 of IXP1200 correspond to the four segments. This implementation covers the data plane functionality as well as the control plane functionality

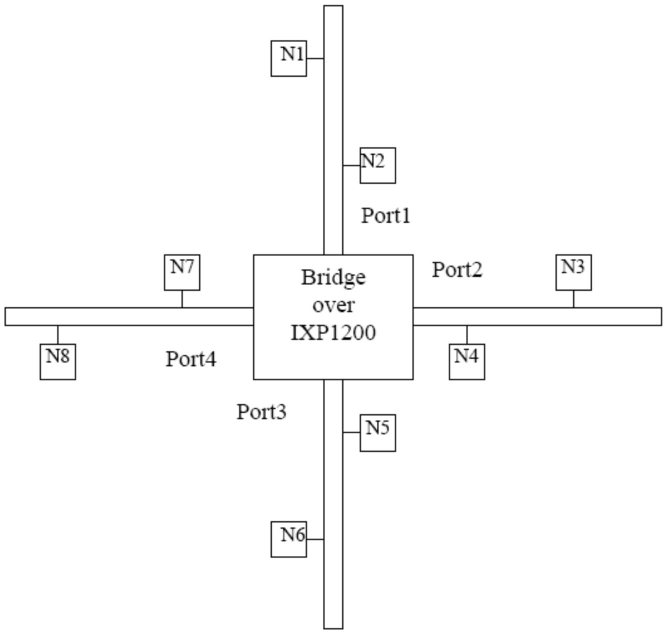

FIG .3: Bridged network

The data plane functionality can be separated into three distinct operations viz. packet reception, Ethernet bridging and packet transmission. The pseudo code for Packet reception is as follows:

Pseudo Code for Packet Reception:

```
Start
If data available at port
        Transfer to RFIFO
        Receive_packet (port num, RFIFO num)
Else keep sensing for data
End if
If for the mpacket, SOP=1,
  Allocate new buffer in SDRAM and transfer packet.
  transfer_mpacket_to_buffer (bufffer_data_ptr, rfifo_num)
Else   If EOP=0 transfer mpacket in buffer
  Else transfer mpacket in buffer
          And buffer free.
End If
End
```

One microengine is dedicated to each of the four ports. It scans the port for availability of data and as soon as 64 bytes of data arrives, it transports this blocks of data to the receive buffer in the IX Bus interface unit known as RFIFO for the respective port. This chunk of data is known as *mpacket*. Each packet has a reference flag to mark the 'start of packet' (SOP) or 'end of packet' (EOP). For an intermediate mpacket both these flags are zero. Further, the packet is transferred to SDRAM memory and reassembled, based on SOP and EOP flags.

Pseudo Code for Data Plane Functionality of Bridging

```
Start
Extract header from the packet
If Control Packet
  Send to Control packet block
Else if Data packet
  Send source Mac address for hashing hash_48( )
  Check hashing table
  If not found, Add_MAC_Address_to_table( )
  Else    hash destination address hash_48( )
    If found, check output interface
      If output interface = input interface
        Discard the packet
      Else send output interface to transmit module
      End if
    Else Set a flag for broadcasting the packet
    End if
  End if
End if
End
```

We extract the header of the packet in 'ethernet_header' structure by typecasting the packet. The source and destination MAC addresses can be directly accessed through this structure. The IXP1200 provides a hashing unit in the IX Bus interface unit. This hash unit can take 48-bit or 64-bit of data, and produce 48-bit or 64-bit hash index respectively. The microengine initiates the hash operation by writing a continuous set of SRAM transfer registers with the Source MAC address used to generate hash index and then executes the hashing function *hash_48( ).* The hash unit uses a hard-wired polynomial algorithm and a programmable hash multiplier to create hash indexes. This saves many instruction cycles which would have been required for software hashing. The hashing table is accessed at the [hash index] location and the output interface is extracted from this table entry. This interface is passed on to the transmit block. If the MAC address is not there, then the add_MAC_address_to_table routine adds the entry.

Pseudo Code for Transmit Module:

```
Start
If flag for broadcast = 0
       get mpacket from SDRAM.
       get_mpacket (pointer to buffer)
Else   get_mpacket_broadcast (pointer to buffer)
End If
If TFIFO Element not validated
  Fill the element with current mpacket
Else Check next TFIFO element
```

```
End If
While transmit_pointer ≠ TFIFO number
    Wait
End while
While Transmit_ready_bit ≠ 1
    Wait
End while
Validate TFIFO element. validate_fifo(fifo_entry)
If last mpacket,
    Buffer free. buffer_free(pointer to buffer)
Else work on new mpacket get_mpacket
End If
End
```

The transmit module takes the pointer to the SDRAM location of the packet as an argument along with the port number where the packet has to be forwarded. It transfers the packet to transmit buffers known as TFIFO s. There are 16 such TFIFO elements in the IX Bus interface unit. A *transmit_pointer* polls these elements in a round robin fashion. When it encounters a TFIFO element which is filled with data, it checks for the availability of buffer space in the forwarding port. When buffer space is available, it validates this TFIFO element. The IX Bus then takes over the control and transfers the packet out through the output port.

We replicate the packet to be broadcasted by manipulating the pointer to the packet in the SDRAM. We restrict the pointer to move until the packet is forwarded to all the output ports. A proper synchronization is maintained between the TFIFO elements and the transmit pointer. If the transmit pointer gets stuck at an unfilled TFIFO element then the code would come to a halt. To avoid this, three out of four threads of a microengine fill the TFIFO elements while the fourth one looks after sending the packet out through the proper port.

Psuedo Code for Control Plane Functionality

```
Start
Hash source MAC address
If source MAC already present at hashed location,
    Reset keep alive timer
Else Store input interface and source MAC address
            at hashing index location
            Add_MAC_Address_to_table( )
Set the broadcasting flag
End
```

The control plane functionality consists of auto-discovery and auto-updates. If a new node attaches to one of the branch, then it sends a control packet to intimate its arrival to other nodes. When the bridge receives this packet, it updates its table and at the same time floods this packet to all ports. This helps all the nodes across the bridge to know that a new node is being added. All nodes send a periodic *keep_alive* control message. If this message is not received by the bridge after fix time from a certain node, the bridge deletes that node from the table. Thus addition and deletion of the nodes is handled.

The implemented code was further optimized for best performance. The microengines were assigned to a specific port of IXP. One microengine was assigned to assist these four microengines in a round robin fashion whenever data is available. The throughput results for this are as follows:

Port	Packets Received	Packet receive rate
0	51	596.21
1	25	292.57
2	25	292.57
3	25	292.57

As can be seen from above result, the spare microengine is assisting the port 0 to have a higher data rate. We have achieved the flexibility to assign variable data rate to any port and the code has been optimized.

The instruction cycles required for each data packet processing were studied. The details are as follows:

Operation	Instruction Cycles
<u>Receiving single Packet:</u>	
Initialization of free list	16
Wait for data at port	17
IX Bus to RFIFO transfer *	73
Get packet info (*rcv_cntl* CSR*)*	20
Buffer allocation	28
Transfer to SDRAM from RFIFO *	51
Update Mpacket Count	04
Increment counters	38
Buffer free	15
<u>Bridge Packet</u>	
48 bit hashing of source	30
checking of hashtable in SRAM *	130
Decision making	05
<u>Transmit Packet</u>	
SDRAM to TFIFO transfer *	75
Polling transmit pointer	30
Checking buffer space at output port	15
Validating the TFIFO element	05
Increment TFIFO entry	03
Total cycles	555

The above table gives the cycles required for bridging operation for data plane functionality. The clock of a single microengine is 200Mhz. So for 555 cycles the time required is 2.775 μs. The entries marked with an asterisk (*) give higher latencies. These latencies were hidden using multithreading. One clock cycle is required for making a context switch. So the effective cycles required for processing of a single packet are calculated as follows: 555-325+5 = 235. This gives processing time of 1.175 μs per packet and a throughput in excess of 500Mbps. The control plane functionality was found to incur a usage of about 30 more instruction cycles. These were required to add the new MAC address to the table. The other functions involved did not access external memory and hence took less instruction cycles. Thus our implementation adds an important functionality to the bridging code, with optimum performance.

The throughput with current parallelism is over 500Mbps.The speed of packet processing can be enhanced by pipelining the operations by assigning the receive functions on one microengine and bridging and transmit functions on another one. One such pair of microengines could be dedicated to each port. Since IXP1200 has only six microengines, currently only 3 ports can be supported. However, IXP2400 has 8 microengines with clock frequency of 600 MHz, which would raise the throughput over 1Gbps on all four ports.

IV. CONCLUSION

We have implemented a scalable model of Ethernet Bridge over IXP1200. The data plane forwarding functionality was demonstrated successfully. The auto discovery and update functionality was implemented over IXP1200 with minimal overhead. The implementation was tested by adding and deleting nodes from the experimental network. The code was optimized for high-sped performance using the multithreading and multiprocessing capabilities of IXP1200. A high throughput was obtained at every port. The code could be further implemented on IXP2400.

REFERENCES

[1] Do-Yeon Kim, Sang-Min Lee, Chang-Ho Choi, Hae-Won Jung, and Yeong-Seon Kim, “Trends of 10 Gigabit Ethernet Switch Development in Korea”.

[2] Andrew Rindos, Steven Woollet, Larry Nicholson and Mladen Vouk, “A performance evaluation of emerging ethernet technologies: switched/high-speed/full-duplex ethernet and ethernet LAN emulation over ATM”.

[3] Johnson T. Kuruvila, S.R. Muthangi, A. Paulraj, “Comparison of Collision Avoidance techniques for efficient voice transmission on Ethernet”

[4] “An Engineering approach to computer networks” , S. Keshav, Addison-Wesley ,1997.

[5] “Computer Networks”, Andrew S. Tanenbaum, PHI, second edition, 1990

[6] “IXP1200 Programming” , Erik Johnson, Aaron Kunze, INTEL PRESS.