



Intel[®] Technology Journal

Network Processors

Network Processor Performance Analysis Methodology

Network Processor Performance Analysis Methodology

Sridhar Lakshmanamurthy, Kin-Yip Liu, Yim Pun,
Larry Huston, Uday Naik
Intel Communications Group, Intel Corporation

Index words: network processors, IXA, OC-48, MEv2, line-rate performance, IP DiffServ, benchmark, IXP2400

ABSTRACT

This paper describes the performance analysis methodology developed to analyze the performance of various networking applications that are targeted for running on the IXP2400 network processor, the second-generation IXA network processor.

Traditionally, CPU benchmarks and system-level benchmarks have been used to understand the performance of general-purpose computer systems. However, such standards are still evolving in the field of network processors. Furthermore, network processors are targeted to diverse applications, and their performance is intricately tied to both the hardware features implemented on-chip and the software running on them, posing significant challenges in developing and using standard benchmarks.

The methodology described in this paper addresses the challenges in analyzing the performance of various networking applications running on the IXP2400 network processor. This methodology involves dividing the application into pipeline blocks, estimating the compute and IO requirements for each block, estimating the available processing and latency budget for each pipeline element, and mapping the application blocks to the software paradigms and the hardware resources. This methodology is validated by writing and tuning the microcode blocks for the application.

This paper also describes a case study using the IPv4 forwarding + DiffServ application running on the IXP2400 to analyze and demonstrate OC-48 line-rate performance for a 46B minimum-sized POS packet.

INTRODUCTION

Network processors are an emerging class of chips that are highly programmable and optimized for processing

packets at wire speed. Specifically, these processors are designed to handle deep packet inspection that spans layers 3 through 7 of the Open Systems Interconnection (OSI) network model and are targeted for applications in the OC1 (55Mb/s) to OC192 (10Gb/s) data rates. Flexibility and programmability make the network processor a good candidate to replace expensive and inflexible ASIC chips for all the fast-path (data plane) processing in network equipment, as these network processors (NPU) can cover a wider range of applications. This provides faster time-to-market, increased flexibility, and lower costs to network equipment Original Equipment Manufacturers (OEMs).

A key challenge in making NPUs successful is establishing their performance capabilities. Traditional CPU benchmarks such as the SPEC CPU2000 [1] suite (comprised of the CINT2000 for integer benchmarks and CFP2000 for floating point benchmarks) have been used extensively to understand the performance of general-purpose CPUs. For benchmarking computer systems, the Transaction Processing Performance Council [2] has developed system-level benchmarks such as TPC-C (On-Line Transaction Processing benchmark), TPC-H (ad-hoc decision support benchmark), and TPC-W (transactional web e-commerce benchmark). However, such standards are still evolving in the field of network processors. Furthermore, network processors are targeted to diverse applications, and their performance is intricately tied to both the hardware features and the software running on them, posing significant challenges in developing and using standard benchmarks. This paper describes a methodology that addresses the challenges involved in analyzing the performance of networking applications running on the IXP2400 network processor and presents a case study using the IPv4 forwarding + DiffServ application.

A key ingredient of the performance analysis methodology is a detailed data movement model of the

target application. This model describes the various operations performed by the network processor on every received packet. Depending on the application, these operations could include protocol header error checks, payload error checks, route lookup, flow identification, complex rule-based filtering, header/payload compression/encryption, policing, congestion management, queuing, scheduling, rate shaping, rate limiting, and transmitting.

The next step of the methodology uses the data movement model to estimate the number of compute cycles and total I/O references required for these operations on a per-packet basis. The total compute cycles are determined by the number of instructions that need to be executed for completing the necessary tasks. The I/O references include all operations to external memories to read and write information required during the processing stage. High-level pseudo-code is developed to arrive at these estimates.

Another key ingredient of the methodology is the estimation of the total available budget for packet processing. This budget is determined based on the packet inter-arrival time, and depends on the network processor frequency, the data rate at which packets are received by the network processor, and the smallest packet size that could be received by the processor.

The results of the above estimation efforts, namely,

1. available budget per-stage based on the processor frequency, data rate and the minimum packet size and
2. total compute and I/O cycles required by the application

determine how the functional blocks are mapped onto the available h/w resources (microengine contexts, on-chip scratch memory, external DRAM and SRAM) and how the software concepts (hyper-task chaining, pool of threads, functional pipeline, context pipeline) are used to meet the performance goals.

The methodology is validated by implementing microcode and tuning the code on the simulator and the hardware to demonstrate line-rate performance.

This paper uses the above outlined methodology to demonstrate OC-48 line-rate performance for 46B POS minimum-sized packets for the IPv4 forwarding + DiffServ application running on IXP2400. The following sections provide a brief overview of the internal and external architecture of IXP2400, describe the data movement model for the IPv4 forwarding + DiffServ application, and discuss the performance analysis and tuning of this application.

IXP2400 NPU OVERVIEW

Figure 1 shows the external interfaces of the IXP2400 NPU. The IXP2400 has two 32-bit interfaces to move network data in and out of the chip. The RX and TX interfaces support industry standard protocols for data movement such as SPI3, POS-PHY-L2 for packet-based interface, Utopia 1,2,3 for cell-based interface, and CSIX protocol for the switch fabric interface [3,4,5].

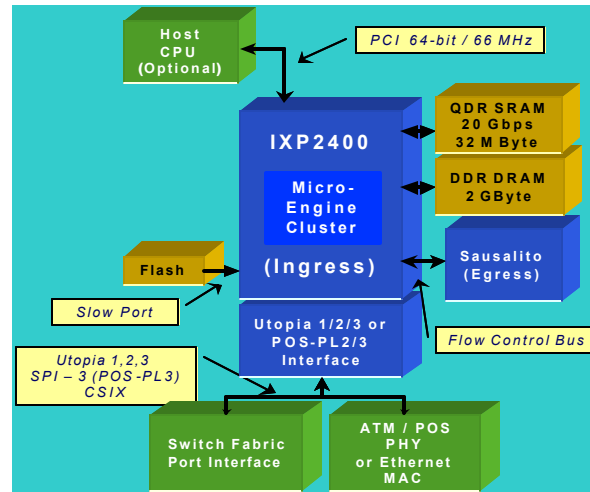


Figure 1: IXP2400 external interfaces

This interface can be independently configured to be 1x32, 2x16, 4x8, or 2x8+1x16 and can be clocked at 25MHz-125MHz, providing full flexibility to use the processor in any application ranging from OC-3 to OC-48 data rates. At 125MHz, the interface provides a peak bandwidth of 4Gb/s in and out of the chip to support the overhead of switch fabric encapsulation.

Since each IXP2400 provides only half-duplex OC-48 connectivity, two such chips are necessary for a full-duplex line card. To support this configuration, the IXP2400 also has a 4b/8b CSIX flow control bus that is used to communicate fabric flow control information between the two processors. At 125MHz, this interface provides up to 1Gb/s of peak bandwidth for flow control messages.

The IXP2400 has one channel of industry standard DDR DRAM running at 150/300MHz, providing 19.2Gb/s of peak DRAM bandwidth. The channel can support up to 2GB of DRAM. The DRAM is primarily used to buffer packets.

In addition to the DRAM, the IXP2400 also provides two channels of industry standard QDR SRAM running at

200/400MHz, providing 12.8Gb/s of read bandwidth and 12.8Gb/s of write bandwidth. Up to 32MB of SRAM can be populated on the two channels. The SRAM is primarily used for packet descriptors, queue descriptors, counters, and other data structures.

The NPU can communicate with the host processor over the 64b, 66MHz PCI. The slow port interface is used to connect to the Flash memory and also provides the general-purpose IO interface.

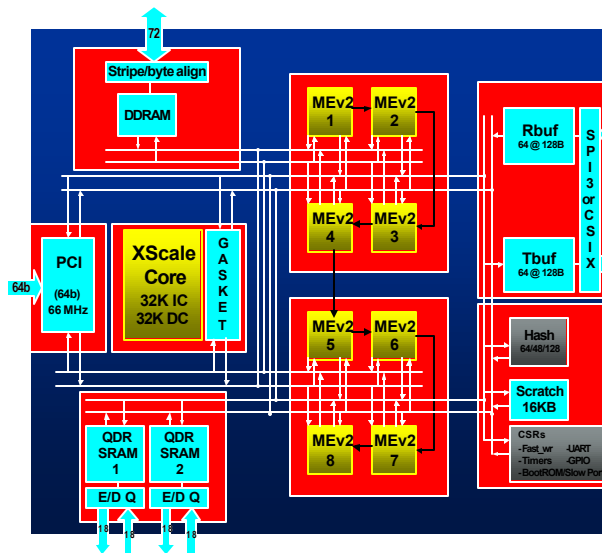


Figure 2: IXP2400 internal architecture

Figure 2 shows the internal architecture of the IXP2400. IXP2400 contains eight multi-threaded, packet-processing microengines. These eight microengines are highly programmable packet processors and support multi-threading of up to eight threads each. Each microengine provides a variety of network processing functions in hardware and provides the ability to process data at OC-48 wire-speed. IXP2400 also offers extensive communication mechanisms between all on-chip processing units and enables the microengines to readily form different topologies of software pipelines that can be customized for various target applications and network traffic patterns. The memory controllers facilitate efficient accesses to the off-chip SRAM and DRAM.

The IXP2400 microengine design includes additional features to increase performance and simplify development. These new features include the following:

- A multiplier for quality of service (QoS) algorithms such as metering and traffic shaping.

- A pseudo-random number generator to accelerate congestion avoidance algorithms like Weighted Random Early Discard (WRED).

Cyclic Redundancy Check (CRC) hardware that verifies and generates Cyclic Redundancy Code (CRC) for Asynchronous Transfer Mode (ATM), ATM Adaptation Layer 5 (AAL5), Ethernet, Frame Relay, and High-Level Data Link Control (HDLC) protocols.

16-entry Content Addressable Memory (CAM) used to implement a data cache in local memory. The CAM facilitates efficient data sharing among microengine threads, resulting in greater performance, as well as reduced consumption of precious memory bandwidth.

A 64-bit local timer with programmable time-out signaling to enhance traffic scheduling and shaping.

640 words (4B) of local memory that is shared by all the threads.

The IXP2400 also has an integrated low-power general-purpose Intel® XScale microarchitecture core. The integrated XScale processor offers ample processing power for running control plane software.

IXP2400 also offers a variety of low-latency communication mechanisms among the microengines and the integrated XScale processor. These communication mechanisms consist of dedicated high-speed data-paths between neighboring microengines, data-paths between all microengines, shared on-chip scratchpad memory, and shared First-In-First-Out (FIFO) ring buffers in scratchpad memory and SRAM. These innovations enable the microengines to form various topologies of software pipelines flexibly and efficiently, allowing processing to be tuned to specific applications and traffic patterns. This combination of programming flexibility and efficient inter-process communication ensures performance headroom while minimizing processing latency.

Network processing applications typically need to perform extensive queue management. Depending on the applications and algorithms used, the network processor may manage thousands of packet queues. The network processor must execute the desired scheduling algorithm and select the appropriate packets out of these queues for transmission at wire speed. As a result, effective queue management is key to high-performance network processing and to reducing development complexity. The IXP2400 provides high-performance queue management hardware that automates adding data to and removing data from queues. Multiple threads can access queues simultaneously. The size of each queue and the number of queues is limited only by the amount of memory available.

Pentium® is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

IXP2400-BASED LINE CARD CONFIGURATION

Figure 3 shows a full-duplex OC-48 line card configuration using the IXP2400. The two IXP2400 processors labeled ingress and egress processors execute the IPv4 forwarding + DiffServ application described in the next section.

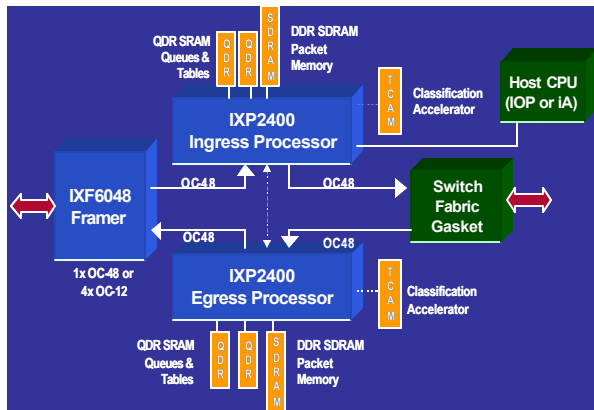


Figure 3: IXP2400-based OC-48 line card configuration

PERFORMANCE BUDGET

A key metric required for the performance analysis is to estimate the available budget. The available compute budget is calculated on a per microengine basis and determines how much processing the network processor can perform on each packet. The available IO latency budget is determined by the number of threads used in a processing block.

The compute budget for a given application is determined by the size of the smallest packet that needs to be processed and the targeted data rate. In other words, the packet inter-arrival time determines how many cycles are available for processing each packet. In order to keep up with the arrival rate, no processing stage in the pipeline can exceed this inter-arrival time.

On the POS interface, the smallest IP packet that can be transferred is 40B (20B of IP header and 20B of TCP header). Assuming a PPP protocol overhead of 6B, the total size of the minimum POS packet becomes 46B. At an

OC-48 data rate of 2.5Gbps, the inter-arrival time between two back-to-back minimum-sized POS packets is 147ns.

The following equation shows the relationship between the various parameters in estimating this packet inter-arrival time:

$$\text{PacketInterArrivalTimeIn(ns)} = \frac{\text{PacketSizeInBytes} * 8}{\text{DataRateInGbps}}$$

The packet inter-arrival time measured in nano-seconds (ns) is obtained by dividing the packet size specified in bits with the desired data rate where the units for the data rate are specified as gigabits per second.

The time in nano-seconds can be converted into processor clocks using the following formula:

$$\text{PacketArrivalTimeInProcessorClocks} = \frac{\text{PacketInterArrivalTimeIn(ns)}}{\text{ProcessorClockTickIn(ns)}}$$

The processor clock tick is the reciprocal of the processor frequency. For example, if a processor is running at 100MHz, the processor clock tick equals 1/100MHz or 10ns. Similarly, 600MHz processor frequency translates to a processor clock tick of 1/600MHz or 1.67ns.

Assuming a 600MHz clock on the MicroEngines (MEs) in the IXP2400, the minimum packet inter-arrival time of 147ns translates to 88 microengine cycles. In order to keep up with the arrival rate, any processing stage of the pipeline must complete all the required processing for a given packet within this budget and should be able to process a new packet every 88 cycles.

The compute cycle requirements for other data rates or minimum-sized packets can be derived in a similar fashion. For example, ATM cells have a fixed size of 53B. At the OC-48 data rate, back-to-back ATM cells arrive every 170ns or 102 microengine cycles. This is the available budget per microengine for processing ATM cells.

Reducing the data rate requirement or increasing the packet size increases the available compute cycles per ME. For example, the total available budget for handling POS minimum packets at the OC-12 data rate (622Mbps) is 355 cycles per ME, four times the available budget compared to OC-48 since the data rate of OC-12 is lower than the OC-48 data rate by a factor of four. Similarly, the above equations can be used to calculate the available budget for handling 100B POS packets at OC-48 data rate. This equals 192 cycles per ME.

Figure 4 shows the relationship between the data rate and the packet size on the available budget per ME in terms of available ME cycles. As indicated above, the available budget increases as the packet size increases or the data rate decreases.

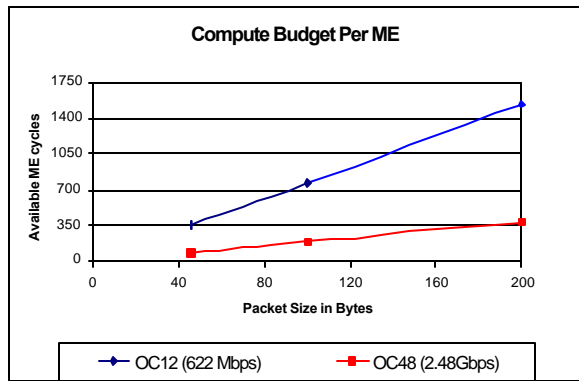


Figure 4: ME compute budget estimate

As the IXP2400 has 8 microengines, the total available compute cycles for the entire application is 8 times the total available budget per ME. In other words, the total available compute cycles on the ingress and egress IXP2400 for the IP DiffServ application at OC-48 data rates for handling minimum POS packets is 8×88 or 704 cycles.

Packet processing involves accesses to internal and external memories such as scratch memory, SRAM, DRAM, etc. The number of memory accesses per stage depends on the data movement model for that stage. Typically, the latency to access memory is several minimum packet arrival times, i.e., SRAM access latency would be 150 cycles (almost twice the POS minimum packet arrival time), while DRAM access latency would be 250-300 cycles (>3 times the POS minimum packet arrival time). The software-controlled multi-threaded features on the ME provide the mechanism to hide these large latencies. Using 8 threads on an ME provides an IO latency budget of 8 times the packet arrival rate. In the above situation of 150-cycle SRAM latency, a total budget of 8×88 or 704 cycles (8 times the POS minimum packet arrival time) allows the stage to support four dependent SRAM operations. Assuming 250-cycle DRAM latency, each stage can support two dependent DRAM operations and still maintain line rate.

In the multi-threaded implementation, each thread within the ME is assigned a new packet that arrives into the system. Assuming M MEs and N threads per ME, a total of $M \times N$ packets can be handled in parallel by these multi-threaded engines before the first thread of the first ME needs to be ready to process the $M \times N + 1$ th packet. This provides a mechanism to increase the total available compute and IO budget.

Typically if M is 1, the software pipeline implementation is referred to as a context pipeline. $M > 1$ implies that several MEs are grouped together to handle a given processing block. Such a pipeline is referred to as a functional pipeline. The following paragraph describes these software pipeline models supported by the IXP2400.

The IXP2000 programming model provides two types of software pipelining models:

A context pipeline, in which different pipeline stages are mapped to different MEs. Each ME constitutes a context pipe-stage. A packet context is passed from one pipe-stage to the next using the various inter-ME communication mechanisms. The available compute budget per context pipeline stage is the same as the budget available per ME.

A functional pipeline, in which a packet context remains within an ME while different functions are performed on the packet as the time progresses. The ME execution time is divided into n pipe-stages and each pipe-stage performs a different function. Multiple MEs are assigned to the functional pipeline to increase the available compute and IO times. For example, if all 8 threads of 4 MEs are used in a functional pipeline, the total compute budget for the minimum POS packet will be $4 \times 88 = 352$ cycles, and the total IO latency budget will be $4 \times 88 \times 8 = 2816$.

The total compute and total IO operations required for a given block determine the pipeline that would be used for that stage.

IPV4 FORWARDING + DIFFSERV APPLICATION

The line card configuration uses the SPI3 [3] interface to receive Internet Protocol (IP) packets from the SONET framer. This mode is also known as the Packet over SONET mode (POS). The fabric interface for the line card uses Common Switch Interface (CSIX) protocol [5], standardized by the Network Processor Forum.

The data movement model definition for this application involves identifying all the processing blocks that are executed for each packet.

The ingress IXP2400 processor receives POS frames that carry IP payload. Since the IXP2400 supports up to 16 logical ports on the framer, the IP packet segments can arrive interleaved. The first pipeline stage reassembles these segments into complete IP packets and stores the

packets into DRAM. In the subsequent pipe-stages, the ingress processor performs the following tasks:

Route lookup to determine the next hop forwarding information by executing a Longest Prefix Match (LPM) algorithm on the destination IP address.

RFC 1812 [6] compliant IP header checks to validate the IP header.

IP packet classification into flows and queues using either a 5-tuple or a 7-tuple lookup, i.e., classification based on IP source and destination address, Transmission Control Protocol (TCP) source and destination ports, protocol field, L2 port, etc.

Execution of a Single-Rate Three-Color Marker (SrTCM) [7] meter pipeline stage to meter the traffic on a per-flow basis and mark the packet as green, yellow, or red, based on the flow parameters and arrival rate.

Execution of a congestion avoidance algorithm such as Weighted Random Early Discard (WRED) that will randomly drop packets when the queue lengths exceed certain thresholds with the goal of minimizing congestion in the fabric.

Another challenge in obtaining OC-48 performance is the ability to add and delete packets from the queue at twice the packet arrival rate and still support a large number of queues. This is achieved on the IXP2400 by using the on-chip high-performance queue management hardware.

The transmit pipeline includes fully programmable schedulers such as Weighted Round Robin to schedule traffic into the fabric, and a transmit engine that adds fabric encapsulation to the IP frame, segments the IP frame into CSIX c-frames, moves c-frame data from memory and

to the transmit buffers, and enables data transmission on the CSIX interface.

The data movement model for the egress processor is constructed using a similar methodology. The first pipeline stage of the egress processor receives the CSIX c-frames and reassembles the original IP payload using the fabric encapsulation information. Subsequent packet processing stages of the egress processor perform further classification, if required, and execute metering and congestion avoidance algorithms. Sophisticated scheduling algorithms such as Deficit Round Robin are implemented on this processor to provide quality of service (QoS) for the network-bound traffic. The final transmit stage of the data movement model segments the IP frame into transmit chunks called mpackets and enables data transmission on the SPI3 interface.

Similar data movement models can be defined for analyzing the performance of other applications such as Asynchronous Transfer Mode (ATM), Segmentation and Reassembly (SAR), ATM traffic management, voice over ATM etc. The next section of this paper describes how this data movement model is used to estimate the cycle count and I/O requirements for each processing stage.

PERFORMANCE ANALYSIS METHODOLOGY

Figure 5 shows the ME allocation and the software pipeline chosen for each of the blocks in the ingress data flow for the IPv4 forwarding + DiffServ application. This partition is based on the cycle count estimates and IO requirements that are derived from the pseudo-code that corresponds to the data movement model described above. The subsequent sections provide the details on the pseudo-code-based analysis and the choice of the pipeline for each of these stages.

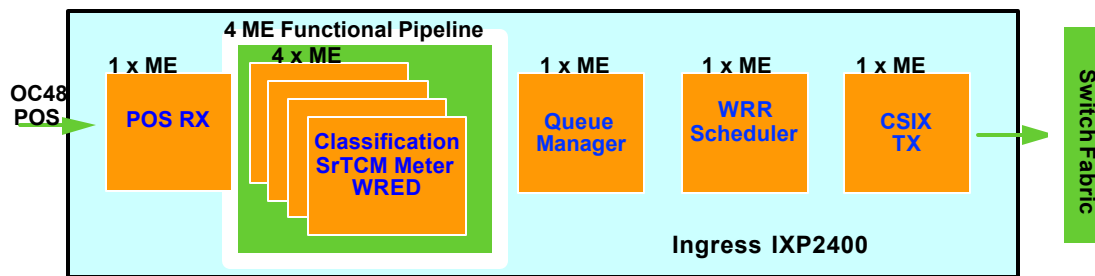


Figure 5: ME allocation for IP DiffServ application

Table 1: Pseudo-code for reassembly stage

Description	Cycle Count	Command	SRAM Rd	SRAM Wr	DRAM Rd	DRAM Wr
The receive thread wakes up, looks at the receive status word (RSW), and decides what to do with the mpacket, based on SOP/EOP indicator within the RSW.	10					
For all conditions (SOP/EOP) move the RBUF data to DRAM. ME updates the reassembly context with the new DRAM location and offset.	30	1				64
For minimum-sized packets (and for EOP), update the buffer descriptor in SRAM	11	1		4		
For EOP, Write SOP ptr, EOP ptr, L2 port, Next IP hop, Class, Payload length to scratch ring for the next pipeline stage	11	1				
Prefetch new buffer descriptor to be used in the next phase	2	1	4			
Clean up, return the thread back to the thread freelist, free RBUF element	6	2				
Total for frame reassembly	70	6	4	4	0	64

POS Receive Block

The first block in the ingress flow is responsible for reassembly of the POS packets. On receiving a new packet, each thread in this block checks the Start of Packet (SOP) and End of Packet (EOP) bits of the packet, identifies the port of the packet, allocates a DRAM buffer for the packet on start of a new packet or when the previous buffer is full, updates the reassembly context with the current offset in the DRAM buffer, moves the data from the receive buffer to the DRAM buffer, signals the next stage of the pipeline on EOP, and cleans up the state for the next round.

This stage requires the following four IO operations: 1) SRAM read to allocate a new buffer; 2) DRAM write to move the packet data into the DRAM buffer; 3) SRAM write to update the packet descriptor information; and 4) a scratch ring write to signal the next pipeline stage when the entire packet has been reassembled with the packet data.

The pseudo-code for this pipeline stage is captured in an excel format and is shown in Table 1. Based on this pseudo-code, the reassembly block for POS minimum packets requires approximately 70 compute cycles and 4 IO operations. Since each ME has an 88-cycle compute budget for handling POS minimum packets, the reassembly function meets the cycle count budget for 1 ME. The next step in the analysis involves estimating the total IP latency for the reassembly block to determine conformance to the 1 ME budget. Assuming 8 threads of the ME are used, the total available IO budget is 704 cycles. Assuming the latency for the SRAM and scratch operations is 125 cycles each, and the latency for the DRAM operation is 250 cycles, the total latency for the 2 SRAM ops + 1 scratch operation + 1 DRAM operation is $2*125 + 1*125 + 1*250$ is 625 cycles. This fits within the 704-cycle available budget.

Thus, based on the pseudo-code analysis and the latency estimate, the POS receive block is implemented as a context pipeline running on all 8 threads of a single ME.

Since the latency estimates used in this analysis are approximate, establishing the actual performance of this block requires successive refinement and tuning of the actual microcode running on the IXP2400-cycle-accurate simulator. This simulation environment provides the dynamic latency value for each of the IO operations based on the total load on the system. The code is tuned such that the dynamic latency budget still fits within the total available IO latency budget.

During the tuning effort, if the dynamic latencies exceed the available budget, other tricks could be used to fit within the budget. The available options include handling multiple packets per thread such that the total available IO budget is further increased. For example, if each thread handles two packets concurrently (with each packet in a different phase of execution), the total available IO budget increases to $2*704$ cycles. However, the flip side of handling multiple packets per thread is the increase in the actual cycle count per thread. Thus, a balance is required to determine whether the compute budget or the IO budget becomes the bottleneck.

In cases where the compute budget exceeds the available cycles on the single ME, the context pipeline stage will be required to be converted into a functional pipeline stage, with additional MEs added to provide the necessary headroom.

Classification Pipeline

The classification stage reads the message from the previous reassembly stage and reads the IP/TCP header of the packet from DRAM. This stage performs routing and classification functions using the information in the header.

One example of packet classification used in the DiffServ application is a 5-tuple exact match lookup to identify the flow and queue ID of the packet. The fields used for the 5-tuple search are IP source and destination addresses, the TCP source and destination ports, and the protocol field. Since the 5-tuple lookup uses 104 bits, a hash scheme is used to efficiently store the necessary information in SRAM. The hash key is generated using the hash hardware on chip. The hash key is used to access SRAM. On completion of the read, the original key is compared to the retrieved key. When a hash collision occurs, multiple dependent SRAM read operations are required to get the flow and queue information.

Once the packet is classified, this pipeline stage also executes all the checks mandated by RFC1812 including decrementing the time-to-live field of the IP header and updating the IP header checksum.

The pseudo-code-based analysis for the classification stage shows that this stage requires 2 DRAM operations plus 9-10 SRAM/scratch operations. Assuming similar latency estimates that are used in the POS Rx analysis, this block requires a total IO latency budget of approximately 1750 cycles, more than the total available IO budget for 2 MEs. Initial pseudo-code-based cycle count estimates showed approximately 160 cycles for performing the 5 tuple exact match and for executing all the header checks mandated by RFC1812, within the budget for a 2 ME functional pipeline.

The SrTCM meter block requires approximately 80 compute cycles and needs only 2 SRAM operations, reads the meter parameters for the given flow, and updates the relevant field of the flow data structure based on the behavior of the current packet. Similarly, the WRED block requires approximately 80 compute cycles and needs only 2 SRAM operations.

Since the classifier block requires more IO budget than the available budget on 2 MEs, while the meter and WRED blocks have a lot of IO budget headroom, the entire pipeline is implemented as a 4 ME functional pipeline, whereby the classification stage can use up the IO budget allocated and unused by the other processing blocks.

Thus, the pseudo-code-based analysis allows the application blocks to be partitioned appropriately to maximize the use of all available resources.

Each thread on each ME handles one packet; thus a total of 32 packets remains active in this pipeline, providing a total latency budget of $4 * 8 * 88 = 2816$ cycles to retire each packet. In other words, each thread receives a new minimum packet every 2816 cycles. In order to keep up with the OC-48 line rate, each thread in the classification stage must retire the previous packet within 2816 cycles.

Queue Manager

The Queue Manager (QM) is responsible for performing enqueue and dequeue operations on the transmit queues for all packets. The functionality of the QM is identical on both the ingress and egress processors: to process enqueue and dequeue requests from the other pipe-stages and perform the necessary operations on the queue array structures. The enqueue operation does not return any data. The dequeue operation returns a pointer to the buffer descriptor that was dequeued. This information is transferred to the ME in the transmit pipeline via a scratch ring. It is possible to request a dequeue from the QM either before a schedule decision is made or after a schedule decision is made. This is implementation specific and depends on the number of queues supported, scheduling algorithm used, etc. On the ingress processor, the dequeue request is issued after the schedule decision, while on the egress processor the dequeue request is issued before the schedule decision.

The pseudo-code-based analysis for the QM shows that each enqueue and dequeue operation requires 4 SRAM/scratch operations each. This stage does not require any DRAM operations. Each thread handles 1 enqueue operation and 1 dequeue operation in parallel; thus the IO accesses required for the enqueue and dequeue operations can be issued concurrently. Thus, the total IO latency is determined by 4 dependent SRAM operations. Assuming 125-cycle latency, this total latency of 500 cycles fits within the latency budget for 1 ME. Thus the QM is implemented as a context pipe-stage running on 1 microengine.

CSIX Transmit Scheduler

The Common Switch Interface (CSIX) scheduler schedules packets to be transmitted to the CSIX fabric. The scheduling algorithm implemented is Round Robin among the ports on the fabric and optionally Weighted Round Robin among the queues on a port. The scheduling and transmit is done a c-frame at a time.

The CSIX scheduler handles

- Flow control messages from the fabric. These messages are sent by the fabric to the egress IXP2400, which sends them on the c-bus to the ingress IXP2400. If the fabric asserts Xoff on a particular Virtual Output Queue (VoQ), the scheduler stops scheduling for the queue.

- Queue transition messages from the queue manager. A queue is scheduled only if the queue has data.

- MSF Transmit State Machine. The scheduler monitors how many packet c-frames are in the pipeline, and if

the number of packets in the pipeline exceeds a certain threshold, the scheduler stops scheduling.

For both the VoQ status and the transmit queue status, the scheduler keeps hierarchical bit vectors and uses the MEv2 Find First Bit Set (FFS) instruction to scan them efficiently. During each loop, the scheduler

Determines if the TX pipeline is within the threshold.

Picks up from where it left off in the last iteration and finds the next bit set and determines which queue to schedule.

Sends a dequeue message to the queue manager to dequeue the head of that queue. The queue manager dequeues a cell (c-frame) from the head of the queue and sends a transmit request on a scratch ring to the CSIX TX microblock.

Pseudo-code analysis of the scheduler block shows that this block is compute intensive. In the worst-case condition, the scheduler exceeds the 88-cycle budget if it has to process fabric flow control messages, process QM queue transition messages, check for MSF transmit count, and schedule a c-frame at each minimum packet arrival slot. However, the worst-case condition is not likely to occur often, and by slowing down the processing task, the scheduler alleviates some of the problems of the worst-case condition, i.e., fabric flow control messages would slow down if the scheduler slowed down the transmit requests to the fabric, and QM queues would build up, resulting in fewer queue transitions from 1->0 or 0->1 if the scheduler falls behind.

The scheduler block has negligible IO operations. Thus, the CSIX scheduler block is implemented as a context pipe-stage executing on a single ME using only 4 threads. One thread of this ME executes the actual scheduling algorithm. Three support threads handle the fabric flow control messages, the QM queue transition messages, and the MSF transmit counter.

CSIX Transmit

The CSIX transmit engine handles the data movement from DRAM to the transmit buffers. This block receives transmit request messages from the queue manager. For each transmit request, a c-frame is transferred into a Transmit Buffer (TBUF), which is then transmitted into the fabric by the MSF transmit state machine.

Every request has an associated packet, which is being segmented into eframes. The associated segmentation state for the packet and the packet meta-data is cached in local memory and is looked up using the CAM. The TX microblock adds the CSIX header onto the c-frame along with the packet data. Along with the CSIX header, a

Traffic Manager (TM) header is also added per c-frame, carrying extra information (destination layer-2 port ID, input blade ID, sequence number, next-hop ID, etc.) about the packet to be passed to the Egress IXP2400. In addition, the flow ID, class ID, input port, and some other fields from the meta-data are passed along to the Egress IXP2400 using a per-packet header pre-pended to the start of the first c-frame of each packet.

Pseudo-code-based analysis of the CSIX transmit block shows that this stage requires 3-4 SRAM/scratch operations plus 1 DRAM operation for scheduling each c-frame. For minimum POS packets, the latency budget is tight. Tricks such as concurrently handling 2 c-frames per thread are used to increase the total IO latency budget such that this block fits as a context pipe-stage that runs on a single microengine.

CONCLUSION

Analyzing the performance of network processors poses major challenges since these chips are targeted to a wide range of applications. A consistent methodology is required to establish the performance capabilities of these processors. This paper described a methodology for analyzing the performance of diverse networking applications that are targeted for the IXP2400 network processor. This methodology involves estimation of the available processing and latency budget per packet, estimation of the total compute and IO operations required per packet for the various pipeline stages of the target application, and mapping the pipeline stages of the application to the available hardware resources on the IXP2400 and utilizing all the available software pipelining techniques to hit the desired performance. The paper also presented a case study using the IPv4 forwarding + DiffServ application to validate the above methodology. This performance analysis methodology can be easily extended to evaluate the performance of other applications running on the IXP2400 network processor or to evaluate the performance of other network processors. The results from this methodology helped establish the performance capabilities of the IXP2400 network processor for various edge and access router applications.

ACKNOWLEDGMENTS

The authors acknowledge the contributions of Mark Rosenbluth, Deb Bernstein, Gil Wolrich, Hugh Wilkinson, Chen-chi Kuo, Eswar Eduri, Muthiah Venkatachalam, and Prashant Chandra.

REFERENCES

- [1] <http://www.spec.org/> Standard Performance Evaluation Corporation.
- [2] <http://www.tpc.org/> Transaction Processing Performance Council.
- [3] <http://www.oiforum.com/> Optical Internetworking Forum.
- [4] <http://www.atmforum.org/> ATM Forum.
- [5] <http://www.npforum.org/> Network Processor Forum.
- [6] <http://www.ietf.org/rfc/rfc1812.txt?number=1812> IETF RFC specifying requirements for IP version 4 routers.
- [7] <http://www.ietf.org/rfc/rfc2697.txt?number=2697> IETF RFC specifying single-rate three-color marker.
- [8] <http://developer.intel.com/design/network/products/npfamily>

AUTHORS' BIOGRAPHIES

Sridhar Lakshmanamurthy is a senior staff architect at Intel's Network Processor Division, focusing on understanding edge/access networking applications, analyzing the performance of Intel's network processor solutions, and defining future enhancements to these solutions. Prior to joining the Network Processor Division, Sridhar focused on platform performance analysis for Intel server chipsets for Xeon & Itanium Product Family (IPF) processors, and system bus specification for the IPF processors. Sridhar joined Intel in 1993 after receiving an M.S. degree in Computer Engineering from Rice University in Houston, TX. He also holds a B.E. degree in Electronics and Communication Engineering from Osmania University, Hyderabad, India. He can be reached via e-mail at sridhar.lakshmanamurthy@intel.com

Kin-Yip Liu co-manages the NPD NPBU Architecture team at San Jose, focusing on network processors for the Access and Edge market segments. His prior assignments include engineering and management positions in the Itanium® Product Family architecture and firmware teams and in the 386SL and 486SL microprocessor projects. Kin-Yip Liu joined Intel in 1990 after completing his Master of Engineering (EE), Bachelor of Science (EE), and Bachelor of Arts (Economics) degrees from Cornell University. Kin-Yip holds four US patents in microprocessor architecture. His technical interests include network processing, computer architecture, and simulator development. His e-mail address is kin-yip.liu@intel.com

Yim Pun is the chief architect for the IXP2400 network processor. His technical interests include network

processor architecture, microprocessor architecture, and network-processor-based system solutions for the switching/routing/ATM/MPLS/IPsec/SSL/VoIP applications in the metro/access/edge markets. He received his Master of Engineering degree from Cornell University in 1987, and a Bachelor of Electrical Engineering-Computer Engineering degree from the University of Wisconsin-Madison in 1986. He can be reached via e-mail at yim.pun@intel.com

Larry Huston is a principal software architect at Intel's Network Processor Division. He is responsible for defining the software requirements for future network processors as well as designing the advanced programming framework. Prior to Intel, Larry was a software architect at NetBoost where he helped design their programming environment for accelerating network applications such as firewalls and intrusion detection. Prior to NetBoost, Larry was a member of the technical staff at Ipsilon Networks where he designed and implemented Ipsilon's protocols for distributed IP switching and forwarding. Larry received his Ph.D. degree in Computer Engineering from the University of Michigan in 1995. He also holds M.S.E. and B.S.E. degrees in Computer Engineering and Aerospace Engineering from the University of Michigan. Larry can be reached via e-mail at larry.huston@intel.com

Uday Naik is a senior staff software engineer at Intel's Network Processor Division. His professional interests include networking, embedded systems, and digital television. Uday received his Master's degree in Computer Science from the University of Indiana, Bloomington in 1992. He also holds a Bachelor's degree in Computer Science and Engineering from the Indian Institute of Technology (IIT), Bombay. Uday resides in San Jose, California, and can be reached via e-mail at uday.naik@intel.com

Xeon™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Itanium® is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Copyright © Intel Corporation 2002. This publication was downloaded from <http://developer.intel.com/>.

Legal notices at <http://developer.intel.com/sites/developer/tradmarx.htm>

For further information visit:

developer.intel.com/technology/itj/index.htm