

Intel® Technology Journal

Network Processors

Packet over SONET: Achieving 10 Gigabit/sec Packet Processing with an IXP2800

Packet over SONET: Achieving 10 Gigabit/sec Packet Processing with an IXP2800

Matthew Adiletta, Donald Hooper, Myles Wilde
Intel Communications Group, Intel Corporation

Index words: network processors, IXP, communication architecture, routing, switching, 10Gbs, Ethernet, ATM, multi-service switches, multi-processors, microprocessor architecture, hardware-based multi-threading, OC-192, OC-48

ABSTRACT

The IXP2800 is the high-end device of a family of network processors developed by Intel Corporation. It is designed for 10 Gigabit/sec data rates, with typical usage in packet forwarding systems. It can be configured with large amounts of dynamic and static storage for buffering hundreds of thousands of packets for up to a million Internet Transmission Control Protocol (TCP) connections. The programmability and parallel nature of this processor chip make it an ideal choice when high performance and ability to quickly adapt to new network standards are the requirements.

This paper describes the evolution of an OC-192 (10 Gigabit/sec) design for Packet over SONET (POS), using the IXP2800. This was a particularly difficult challenge due to the fact that the arrival rate of packets is one per 40nS (which equals 57 microengine processor cycles). At this rate, for each packet, buffers must be allocated, the packet must be received, reassembled, and stored in DRAM, header verified, multi-field and destination lookups performed, packet classified for destination, timestamp saved, packet tagged (metered) by priority, previous header stripped, IP header modified, evaluated as to whether it should be dropped, statistic counters updated, enqueued for transmit, new fabric header prepended, transmitted, and buffers freed. This paper chronicles the issues encountered and solutions devised to achieve high packet-forwarding rates. The resulting architectural concepts of context pipe stages, functional pipe stages, phase interleaving, critical sections, elasticity buffers, and pool of threads are defined. The techniques of next-neighbor message passing, scratch rings, signaling, CAM state caching, local memory link-lists, SRAM link-lists, and reflected mailbox messages are explored. The OC-192 POS pipe stages are described.

A performance summary is reported for a variety of packet streams.

INTRODUCTION

The IXP2800 started with a simple marketing requirement: Achieve a high-end device that supports 10 Gigabit/sec data forwarding rates and is scalable to much higher rates when configured with a switch fabric.

In parallel with developing the hardware architecture, the IXP2800 design group decided also to develop a proof-of-concept application to explore how programs could be developed on this chip. Packet over SONET was chosen, as this presented the most difficulty in achieving full wire rate. When used over SONET, the minimum IP packet size is 40 bytes. A PPP layer 2 encapsulation is used with this, which adds another 9 bytes (4-byte header, 5-byte trailer), for a total 49B minimum packet size. By comparison, the minimum packet size for Ethernet is 64 bytes with an interpacket gap of 20 byte times.

We assume the reader is familiar with the Intel Network Processor Family [1].

CHALLENGES AND SOLUTIONS

Several obstacles had to be overcome to achieve the required performance. The following sections describe each problem and a corresponding solution.

Context Pipe Stage

Some of the operations on packets are well defined, with minimal interface to other functions or strict order implementation. Examples include update-of-packet-state information, such as the current address of packet data in a DRAM buffer for sequential segments of a packet, updating linked list pointers while enqueueing/dequeueing

for transmit, and policing or marking packets of a connection flow. In these cases the operations can be performed within the 57-cycle stage budget. Further difficulty arises in keeping operations on successive packets in strict order and at the same time achieving cycle budget across many stages. A block of code performing this type of functionality is called a *context pipe stage*.

In a context pipeline, different functions are performed on different Microengines (MEs) as time progresses and the packet context is passed between the functions or MEs. Each ME constitutes a context pipe stage (Figure 1). Cascading two or more context pipe stages constitutes a context pipeline. The name *context pipeline* is derived from the observation that it is the context that moves through the pipeline.

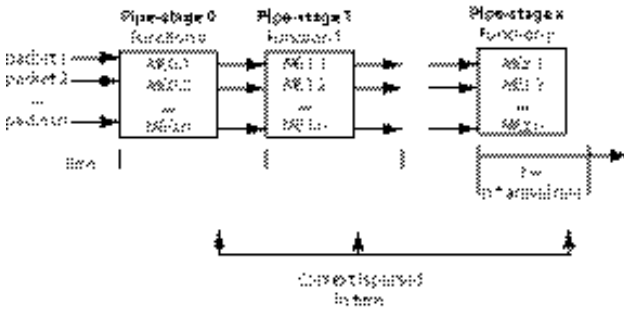


Figure 1: Context stages

A context pipeline is ideally suited when either the program store required to maintain this pipe-stage pipeline function is large or the data set associated with this function is large, or both. Additionally, it is appropriate when the data set associated with the function is greater than the data set associated with the context. An example is the data set for a transmit scheduler: keeping in the local memory up-to-date prioritized link-lists containing state information for active transmit queues. Such state information could include queue cell counts and flow control status.

Storing the information locally enables a function to efficiently maintain state, whereas the context for this pipe stage concerns which queue is being tasked with transmit. In this case it is obvious that the queue number (context) is much less than the data associated with the link lists (function data set).

Another advantage of the context pipeline is that the entire ME program memory space can be dedicated to a single function. This is important when a function supports many variations that result in a large program memory footprint. Cases in which the context pipeline is not desirable are ones in which the amount of context

passed to and from the pipe stage is so large that it affects system performance.

Each thread in an ME is assigned a packet, and each performs the same function but on different packets. As packets arrive, they are assigned to the ME threads in strict order. There are eight threads typically assigned in an IXP2800 ME context pipe stage. Each of the eight packets assigned to the eight threads must complete its first pipe stage within the arrival rate of all eight packets.

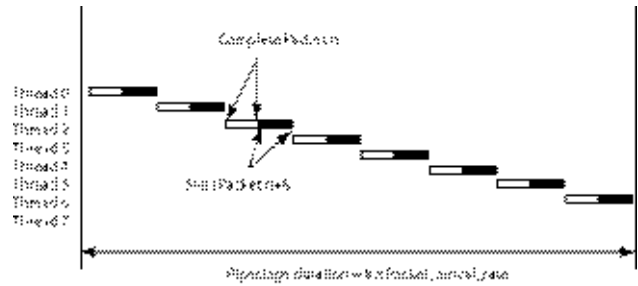


Figure 2: Interleaved phased piping

A more advanced context pipelining technique is shown in Figure 2. This technique interleaves multiple packets on the same thread, spaced eight packets apart. An example would be ME0.1 completing pipe-stage 0 work on packet 1, while starting pipe-stage 0 work on packet 9. Similarly, ME0.2 would be working on packet 2 and 10. In effect, 16 packets would be processed in a pipe stage at one time. Pipe-stage 0 must still advance every 8-packet arrival rates. The advantage of interleaving is that memory latency is covered by a complete 8 packet arrival rate.

Functional Pipe Stage

Another problem arises when the size of the packet state information passed between functions is so large that it is very costly (in cycles) to pass this information between MEs.

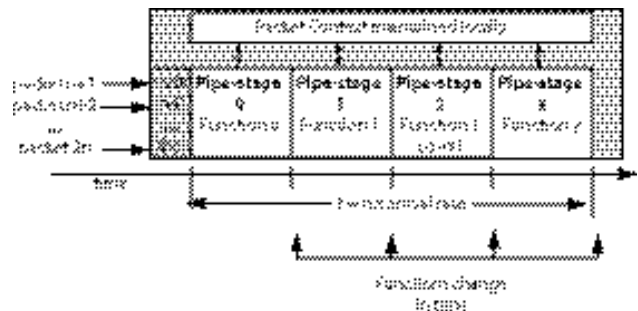


Figure 3: Functional stages

Context pipes can also be wasteful of instruction cycles if the functions being implemented are extremely simple.

In these cases, blocks of code can be arranged in what is known as a *functional pipeline*, a name derived from the observation that functions move through the pipeline.

In a functional pipeline, the context remains with an ME while different functions are performed on the packet as time progresses. The ME execution time is divided into n pipe stages, and each pipe stage performs a different function. As with the context pipeline, packets are assigned to the ME threads in strict order.

There is little benefit to dividing a single ME execution time into functional pipe stages. The real benefit comes from having more than one ME execute the same functional pipeline in parallel. Figure 3 shows four functional pipe stages distributed across 4 MEs, each ME executing with eight threads.

A packet remains with a thread for a longer period of time as more MEs are added to the functional pipe stage. In this example, the packet remains with a thread-32 packet arrival time (8 threads x 4 MEs) because thread ME0.0 is not required to accept another packet until all the other threads get their packets.

The number of pipe stages is equal to the number of MEs in the pipeline. This ensures that a particular pipe stage executes only in one ME at any one time. This is required to provide a pipeline model that supports critical sections. A critical section is one in which an ME thread is provided exclusive access to a resource (such as CRC residue, reassembly context, or a statistic) in external memory. Critical sections are described in more detail later.

Functions can be distributed across one or more pipe stages; however, the exclusive access to resources as

described above cannot be supported in these pipe stages.

The goal when designing a functional pipeline is to identify critical sections and place them into their own pipe stages. The non-critical sections of code then naturally fall into pipe stages that become interleaved with the critical sections. Non-critical code that takes longer than a pipe-stage time to execute must be allocated to more than one pipe stage.

The advantages of functional pipelines are these:

1. Unlike the context pipeline, there is no need to pass the context between each pipe stage, since it remains locally within the ME.
2. Functional pipelines support a longer execution period than context pipe stages.

The disadvantages of functional pipelines are these:

1. The entire ME program memory space must support multiple functions.
2. The latency of the functional pipeline is fixed at the worst-case path through any sequence of functions. If this time is exceeded, the packet must either be dropped or handed off to a slow path (e.g., to Xscale™) to complete packet processing.

Mixed Pipelines

Mixed pipelines employ elasticity buffers between pipelines. Figure 4 shows a context pipeline feeding a ring elasticity buffer that feeds a functional pipeline. The Functional pipeline subsequently feeds another ring elasticity buffer and finally a context pipeline.

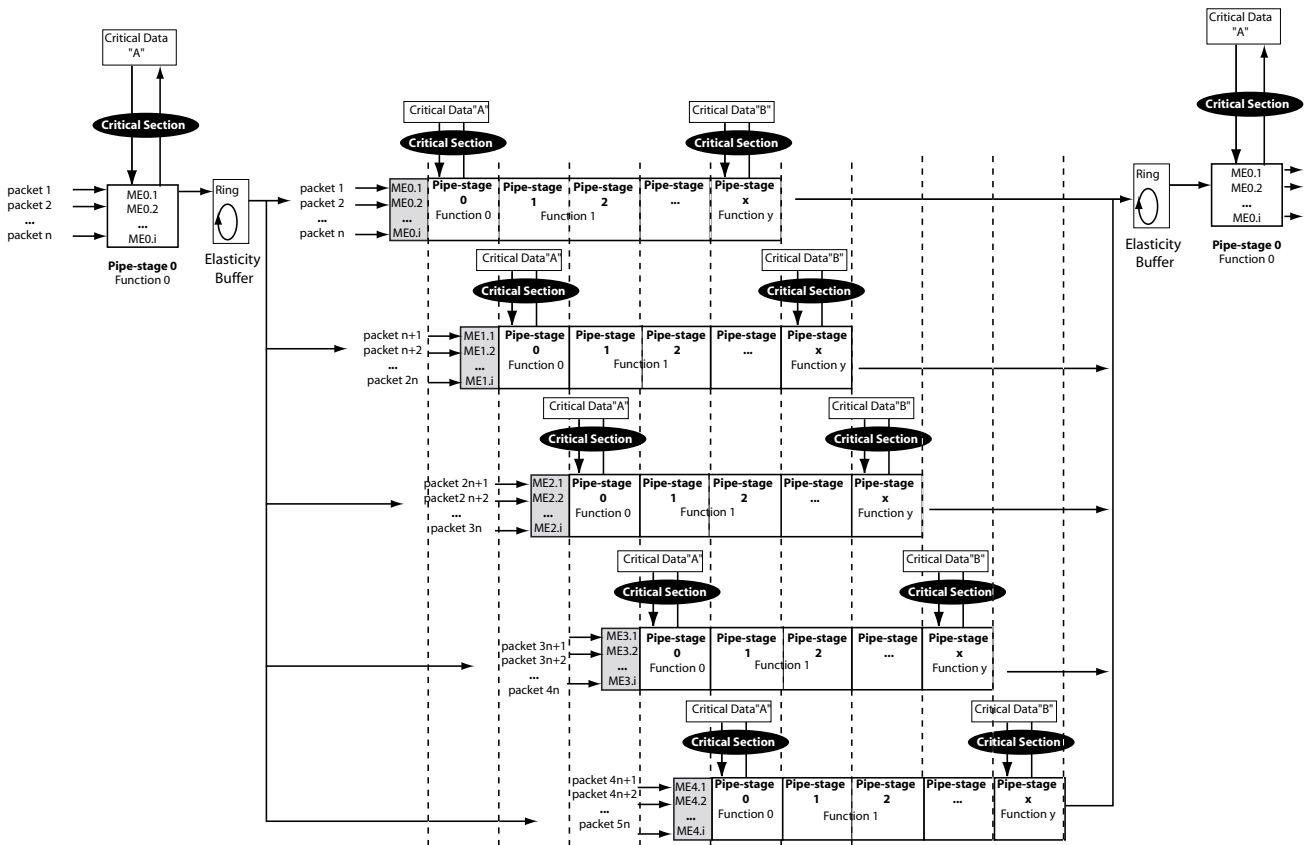


Figure 4: Ring elasticity buffers

Elasticity Buffers

When a pipeline transition occurs, pipeline lock-step execution is not required, and an elasticity buffer (implemented as a ring) can be used. Each pipeline must keep up with line rate. Each pipeline may have multiple pipe stages. The pipeline writing the ring is the producer. The pipeline reading the ring is the consumer. The IXP2800 has hardware assist for maintaining producer consumer rings. The problem to be solved is that there may be single or many producers, and single or many consumers and packet order for both producer and consumer must be maintained. The hardware assist includes hardware-maintained head and tail pointers for multiple rings in either scratch or SRAM memory. When a single producer communicates with a single consumer, Next Neighbor Register Rings can be employed. This is a very low latency private data path.

A pipeline that contains many producers (a multi-ME functional pipeline) can use software techniques in order to maintain order. MEv2 provides a Generalized Thread Signaling (GTS) mechanism for optimized order maintenance among the threads (defined below).

The advantage of using elasticity buffers is twofold. By decoupling producer/consumer heartbeats, independent

software development teams can develop different pipe stages independently and connect them using an API to the ring. Secondly, ring buffers allow short-term line rate processing anomalies/jitter to be hidden.

Synch Section Signaling and Critical Signaling

In a functional pipeline, an ME should not transition into a critical section pipe-stage unless it can be assured that the previous ME has transitioned out of that critical section. In addition, the previous critical section must make sure its write data has progressed such that the next pipe stage reading the data gets the latest coherent data. This can be accomplished by placing a fence around the critical section using inter-thread signaling. There are four ways to signal another thread using inter-thread signaling, as listed in Table 1.

Table 1: Interthread signaling methods

Thread-Signaling Method	Mechanism
Signal next thread in the same ME	Local csr write
Signal a specific thread in the same ME	Local csr write
Signal the thread in the next or previous ME	Local csr write
Signal any thread in an ME	CSR write to CSR Access Proxy Unit (CAP)

Synch Sections

A synch section is a write-dominated section of code in which multiple writers to the shared resource must be sequenced correctly. This is the feather-passing problem. In order to provide strict thread sequencing, a feather is passed from one thread to the next, and a thread can only begin execution when it owns the feather (Figure 5). The IXP2800 uses General Thread Signalling (GTS) to pass the feather. An example of a synch section is code that writes to a ring for enqueue requests while maintaining packet order. The next thread does not have a read conflict with the prior thread, but it must not write to the ring before the prior thread completes its write.

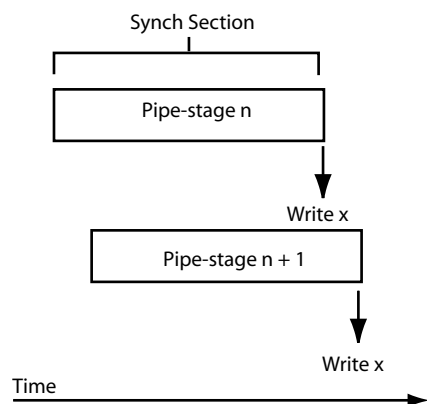


Figure 5: Synch section

Critical Sections

A critical section is a section of code that has only one ME thread with exclusive modification privileges for a global resource (such as a location in memory) at any one time (Figure 6). This is to protect coherency during read-modify-write operations. The following discussion focuses on providing exclusive modification privileges

between the MEs and providing exclusive access between the threads in an ME.

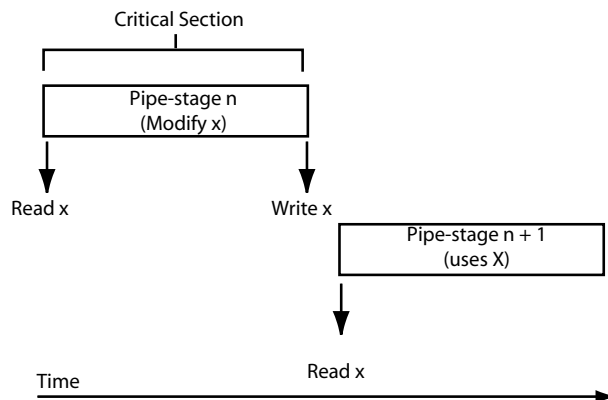


Figure 6: Critical section

Exclusive Modification Privileges between MEs

To ensure exclusive modification privileges between MEs, the following requirements must be met.

- Requirement 1: Only one function modifies the critical section resource.
- Requirement 2: The function that modifies the critical section resource executes in a single pipe stage.
- Requirement 3: The pipeline is designed so that only one ME executes a pipe stage at any one time.

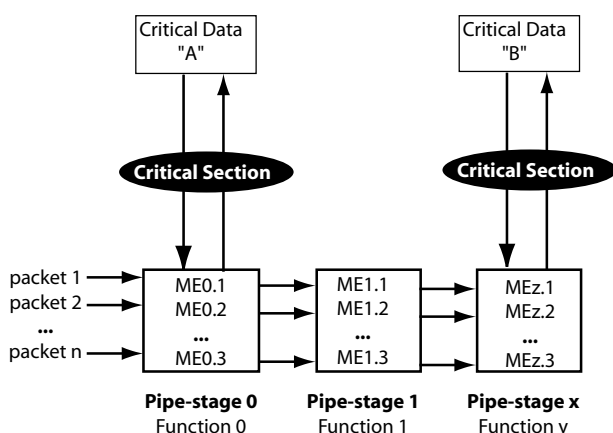


Figure 7: Context pipe with critical sections

Figure 7 shows a context pipeline that supports two critical sections. Each ME is assigned exclusive modification privileges to the critical data, satisfying requirement 1. Requirements 2 and 3 are satisfied because each pipe stage is partitioned into different functions and only one ME executes a specific function.

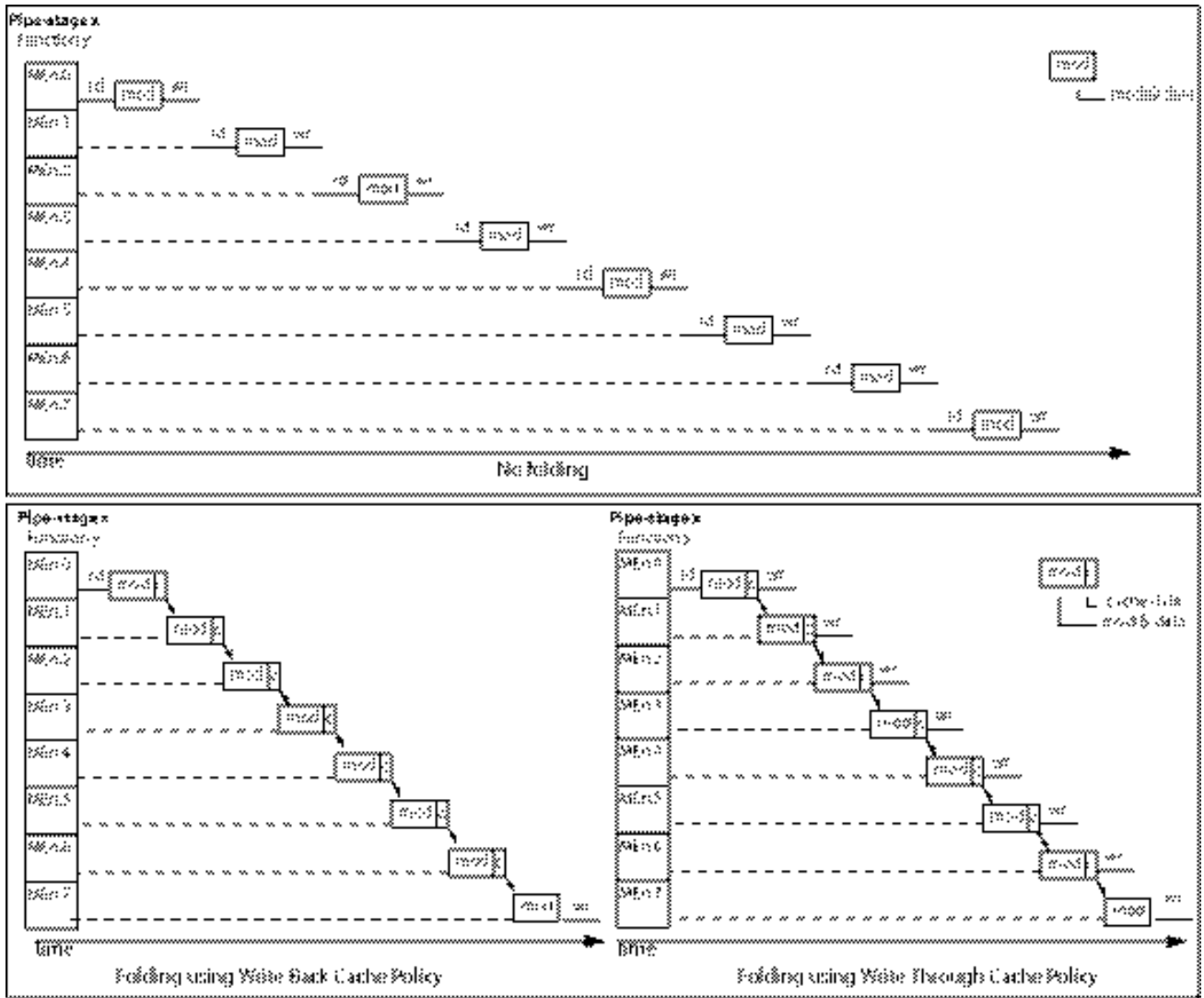


Figure 8: Folding

Folding – Exclusive Modification Privileges between Threads in an ME

Figure 4 also shows a 5 pipe-stage functional pipeline that supports two critical sections. Each pipe stage is assigned exclusive modification privileges to the critical data, satisfying requirement 1. Requirements 2 and 3 are satisfied in the cases shown in the figure; however, it should be noted that a critical section could not be supported by function 1 because it does not satisfy requirements 2 and 3; therefore, two MEs could access the critical data at the same time.

A critical section involves three steps:

1. Reading a resource.

2. Modifying the resource.
3. Writing back the modified data.

As shown in Figure 8, if more than one thread in a pipe stage is required to modify the same critical data, a latency penalty will be incurred if each thread reads the data from external memory, modifies it, and writes the data back. To reduce the latency penalty associated with the read and write, the ME threads can use the MEV2 Content-Addressable-Memory (CAM) to fold these operations into a single read and one or more modifications

A context pipe stage is the only ME that uses the critical data. Therefore the replacement policy for CAM entries is to replace the LRU only on CAM misses. Functional pipelines perform the same function in multiple MEs and therefore are required to evict all the critical data to

external memory before it exits the critical section pipe stage. It should also ensure that the CAM is cleared at the beginning of the pipe stage before any of the threads use the CAM.

Before a thread reads the critical data, it searches the CAM using a critical data identifier such as a memory address. The search will result in one of three possibilities.

1. Miss: When the result is a CAM miss, the critical data is not saved locally, and the thread must read it from external memory. The miss status also includes a Least Recently Used (LRU) CAM entry number.

An ME that executes a context pipe stage will be the only ME that uses the critical data. Therefore, the CAM replacement policy is to replace the LRU only on CAM misses. So the first thing a context pipe stage does on a CAM miss is to evict the LRU data from the local memory back to external memory. Functional pipelines perform the same function in multiple MEs and therefore are required to evict all the critical data to external memory before it exits the critical section pipe stage. Therefore, it can be assured that on CAM miss, the data will already have been evicted. (Note: It is a good programming practice for a functional pipe stage to ensure that the CAM clear instruction is executed at the beginning of the pipe stage before any of the threads use the CAM).

The ME thread then locks the CAM entry and issues a read to get the new critical data. The lock is asserted to indicate to other threads that the data is in the process of being read into local memory. Once the critical data is returned, the thread processes the data, makes any modification to the data, writes the critical data into local memory, and then unlocks the CAM entry.

2. Lock: When the result is a CAM lock, another ME thread is in the process of reading the critical data, and that thread should not attempt to read the data. Instead, it should test the CAM at a later time and use the data when the lock is removed

3. Hit: When the result is a CAM hit, the critical data resides in local memory.

In all cases, the ME thread is assured exclusive access to the data by performing the modification and write operations on the critical data without swapping out.

Pool of Threads

One of the problems with a functional pipeline is that it has a fixed maximum latency, at which time the packet must either be dropped or handed off to a slow path for processing. To get around this, the pool-of-threads concept is proposed.

In place of the functional pipe are 3 blocks:

1. A dispatcher, which assigns packets to available free processing threads.
2. A pool of threads. Each thread makes itself available to the dispatcher for processing tasks. The thread receives a task from the dispatcher, performs the functions of the functional pipeline, then writes results to a re-synchronizing ring.
3. An Asynchronous Insert Synchronous Read (AISR) ring for reordering packets and passing control to the next stage of pipeline. This replaces the elasticity buffer ring.

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

INGRESS: IP PACKETS TO CSIX

This section describes the implementation of the 10Gb/sec Packet over SONET design. Table 2 lists seven fundamental tasks performed in the ingress processor. Figure 8 illustrates these tasks mapped to MEs.

Table 2: POS ingress fundamental tasks

Task	Stage Type	
Frame Assembly Classification	Functional	Performs cell and frame reassembly. Performs IP destination and 5-tuple classification. Maintains the individual data flow contexts. ATM AAL5 CRC checking.
Meter/policing	Context	Single-Rate Tri-color Marker ACLs and flow-based policing.
Congestion management	Context	WRED.
Statistics	Context	Updates statistics counters.
Transmit scheduler	Context	Schedules dequeue for transmit operations. 4-class round robin.
Queue manager	Context	Performs both enqueue and dequeue functions.
Transmit data	Context	Simple transmit to Media Switch Fabric Unit (MSF). buffer management. CSIX segmentation.

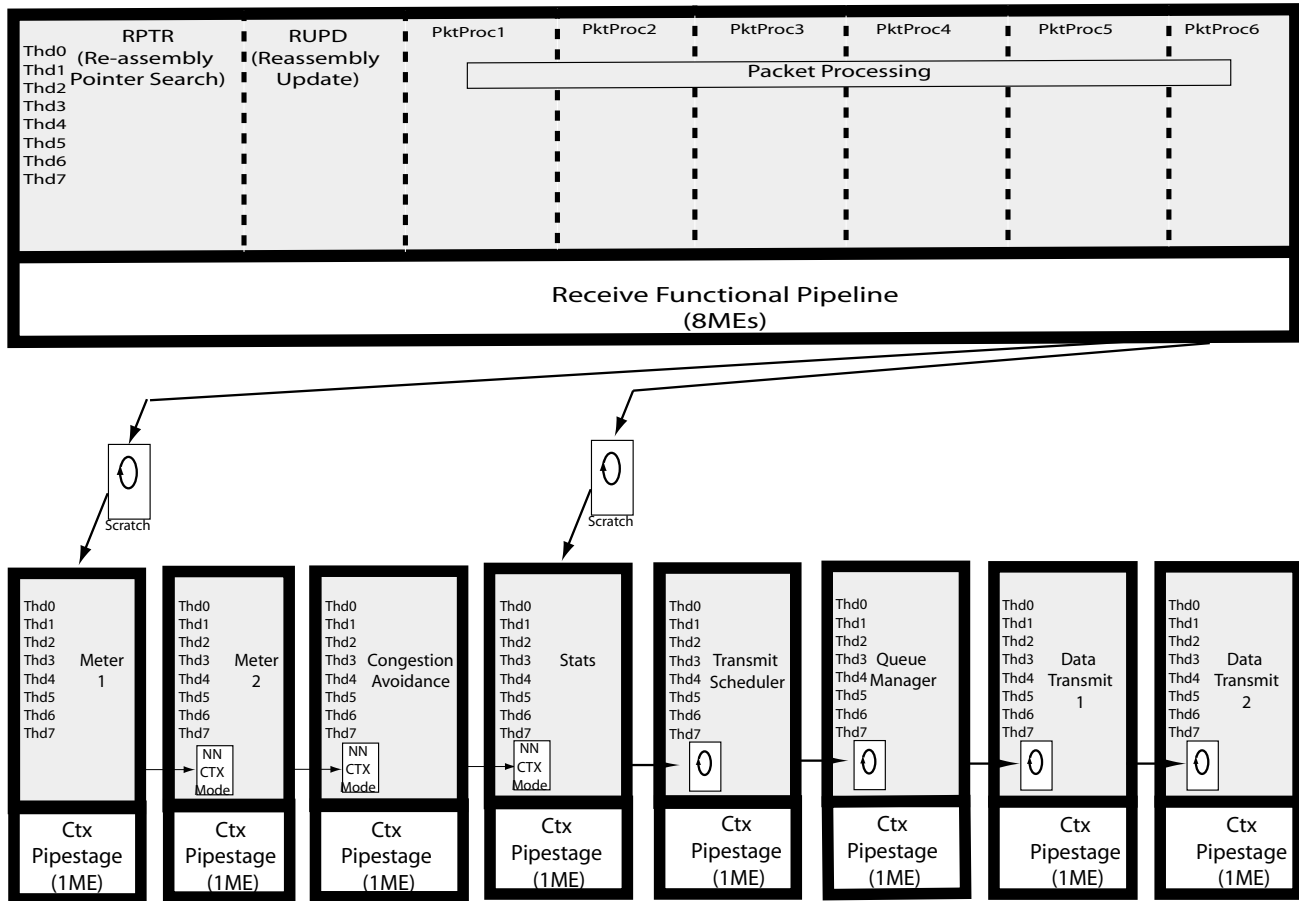


Figure 9: Proof of concept stages

Reassembly Pointer Stage (RPTR)

The RPTR pipe stage finds the pointer to the reassembly state information in SRAM so that the Reassembly Update (RUPD) critical section can perform its read-modify-write on the data as fast as possible. The SPI-4 and CSIX interfaces segment packets into smaller SPI-4 and CSIX frames. The RPTR, RUPD, and packet processing pipe stages work together to reassemble these segmented frames back into full packets. The reassembly state information is used to keep track of the reassembly process.

The RPTR pipe stage also looks at the L2 header, to determine the offset into the IP header. This offset is used by the next pipe stage (RUPD) during reassembly. It is not included in the RUPD because the RUPD is a critical section.

Reassembly State Update Stage (RUPD)

RUPD pipe stage is a critical section that provides the packet processing pipe stage with a pointer to the location in DRAM where the network data should be assembled. It is a critical section because it modifies the

reassembly state information that is maintained in SRAM. The RUPD pipe stage manages this data structure exclusively. Managing the reassembly state involves allocating buffers and calculating offsets, byte counts, and other variables. The MEv2 CAM is used to maintain a coherency of the reassembly state between the eight threads. It is important to note that if the RBUF Control indicates that the complete packet is in the RBUF element, the reassembly information does not need to be accessed and updated. This greatly reduces the memory system bandwidth for packets of 128 bytes or less.

Packet Processing (PPR)

The packet processing pipe-stage threads complete the reassembly process by writing the data to the buffer and also looking at the L2 through L7 packet headers to process the packet. This pipe stage is very application dependent and is expected to change from one application to another. The initial reference design will support the following:

- Stripping the PPP header from the packet.

- Verifying the IP header.
- Determining the destination port with an IP destination search.
- Identifying flows and support access lists with a 7-tuple search.
- Modifying the IP header TTL and checksum.
- Appending a new level-2 header for the packet, containing the classification state to be used by the egress processor.

The packet processing pipe stage is *not* a critical section. It executes six times longer than the other pipe stages in the functional pipeline. The packet processing pipe stage ends with a synch section to ensure that packet order is maintained.

Metering 1 and Metering 2

The Single-Rate Three-Color Marker (srTCM) meters an IP packet stream and marks its packets either green, yellow, or red. Marking is based on a Committed Information Rate (CIR) and two associated burst sizes: a Committed Burst Size (CBS) and an Excess Burst Size (EBS). A packet is marked green if it doesn't exceed the CBS, yellow if it does exceed the CBS but not the EBS, and red otherwise [2].

Metering is performed when an EOP is received for a packet. The packet processing pipe stage performs a flow identification search and gets a pointer to the metering information. This pointer is passed to the metering functions. Metering is performed in two pipe stages (Meter 1 and Meter 2). The first stage reads the pointer from a scratch ring, reads the metering parameters, and calculates the number of tokens collected. The second metering pipe stage performs the actual metering function and writes the updated meter information back to memory. Note that the two pipe stages access and use the same metering parameters. The MEv2 CAM in both pipe stages is used to maintain coherency of the parameters between the threads with an ME and between the two pipe stages. This is possible because the ME threads process packets in strict order and any CAM hit in the first pipe stage is guaranteed to be in the CAM of the second pipe stage. Note that the reverse is not true since the first pipe stage will be working on eight new packets while the second stage is processing its pipe stage.

Congestion Avoidance

Congestion avoidance techniques monitor network traffic loads in an effort to anticipate and avoid congestion at common network bottlenecks [1]. One of

the ways in IP QoS is through packet dropping¹. This requires a way to estimate the average queue size on each packet arrival. Basically, after the packet is classified and before it is enqueued, this block looks at the average size of the queue, compares it to certain thresholds, and makes a decision on accepting or dropping the packet.

RED

The Random Early Detection (RED) was originated from S. Floyd and V. Jacobson [3]. According to their paper, the gateway detects incipient congestion by computing the average queue size. The gateway could notify connections of congestion either by dropping packets arriving at the gateway or by setting a bit in packet headers. When the average queue size exceeds a preset threshold, the gateway drops or *marks* each arriving packet with a certain probability, where the exact probability is a function of the average queue size.

RED gateways keep the average queue size low, while allowing occasional bursts of packets in the queue. During congestion, the probability that the gateway notifies a particular connection to reduce its window is roughly proportional to that connection's share of the bandwidth through the gateway. RED gateways are designed to accompany a transport-layer congestion control protocol such as TCP. The RED gateway has no bias against bursty traffic and avoids the global synchronization of many connections, decreasing the connections' window at the same time.

WRED

WRED combines the capabilities of the RED algorithm with flow-specific DSCP and associated RED parameters. This combination provides for preferential traffic handling for higher-priority packets. It can selectively discard lower-priority traffic when the interface starts to get congested and provide differentiated performance characteristics for different classes of service.

Statistics

The ingress processor supports statistics for incoming traffic, while the egress processor supports statistics for traffic from the switch fabric. Statistics for the flow based on the 7-tuple lookup are also kept for packets_transmitted and packets_dropped. This section applies to both the ingress and egress processors.

¹ Floyd and Jacobson [3] suggest marking instead of dropping. The congestion and ECN-capable bit are only recently being defined in RFC 2481 as an experimental protocol.

RFC2863 [4] states that 64-bit counters must be supported for interfaces that operate at data rates greater than 650Mbps; otherwise, the rollover rate will be too frequent for a monitor program to maintain. Although the standard states that 64-bit statistics are required, it is not absolutely necessary since the rollover rate is 468 years. The POS Proof of Concept implementation uses 63-bit statistics that roll over once every 234 years.

The POS Proof of Concept design includes various assumptions concerning statistics:

1. There are two separate types of statistics maintained for data arriving from the media interface: those associated with an IP interface and those associated with a flow.
2. The IP interface statistics are limited to one set per physical port.
3. Each of the flow and IP interface statistics requires one byte count, one packet count, and one timestamp indicating the last time something was received.
4. Both the flow and IP interface statistics may require 63-bit statistics.

Transmit Scheduler

The transmit pipeline begins with the transmit (Tx) scheduler context pipe stage. The Tx scheduler schedules packets to be transmitted to the CSIX fabric and passes an enqueue and dequeue request onto the next pipe stage (queue manager). The transmit scheduler is also responsible for maintaining the queue status state.

The scheduling algorithm is implemented in software and can be modified per customer requirements. The assumption made for the example design is that the scheduler is transmitting into a switch fabric that supports 16 line cards, each line card having eight ports, and that each port on a line card supports four class types. The total number of ports in the system is 16 x 8 or 128. The example design implements a hierarchical scheduling algorithm. There are four class (priority) wheels, each with 128 entries. The 128 entries each represent a port. Each class wheel is serviced with a Round Robin (RR) scheduling algorithm. A programmable distribution wheel is used to select which ring gets service at each scheduling interval. The programmable distribution wheel provides an anti-starvation mechanism that ensures fairness with some degree of programmability. The example design uses a work-conserving algorithm (it searches for work to do during each scheduling time). If no work is found on a particular class wheel, then the search moves to the next wheel entry.

The transmit scheduler is made up of two threads: schedule (Thread 0) and flow control (Thread 2). Note that the scheduler ME is configured to operate in 4 context mode; therefore, the four threads are numbered 0,2,4,6. Threads 4 and 6 of the scheduler ME are unused.

The scheduler thread uses a singly linked list in local memory (LM) to determine the next eligible schedule for each class. Each class wheel has a 32-bit control register that is stored in LM. This control register is made up of a 16-bit previous queue pointer and a 16-bit current queue pointer that maintain the active links for the class wheel. Each link list entry is made up of an 8-bit next queue pointer, 23-bit queue cell count, and 1-bit flow control status that is stored in LM.

Enqueue information is passed from the receive pipeline through the statistics ME to the scheduler ME. The statistics ME computes the number of cells that each enqueue packet contains from its enqueue state. The statistics ME encodes the computed cell count in one of four enqueue words that are passed to the scheduler ME via a Next Neighbor (NN) FIFO. The scheduler ME uses the cell count information to maintain the queue status for each queue.

Queue Manager

The queue manager is responsible for performing enqueue and dequeue operations on the transmit queues for all packets. Although IXP2800 can support either a linked list or ring queue structures, this implementation of the queue manager is designed to support the linked list queue structure.

Transmit 1 and Transmit 2

The transmit data pipe stages receive a transmit request from the queue manager and, in response, segment the buffer packet data into c-frame segments and moves the data into a TBUF so that the MSF transmit state machine can send the data to the fabric. When all the data in a buffer is transmitted, it frees the buffer by placing it onto a buffer free list.

The queue manager places transmit requests onto a next-neighbor ring. The transmit data pipe stage reads the request and processes the request.

To process the request, the transmit data thread must first check the CAM to see if the buffer descriptor is cached in local memory. If not, it evicts the LRU buffer descriptor to SRAM and reads in the new buffer descriptor. When the buffer descriptor is local, the transmit data thread checks to see if it is an SOP. If so, it finds the current packet data location and offset into the buffer and reads the IP header into transfer registers,

updates the IP TOS field with the DSCP, and updates the IP header checksum. Then it submits the data to the MSF for transmit, with the following steps:

1. Writes the IP header and the CSIX L2 A and B headers to the TBUF element data.
2. Moves the remaining data from the DRAM to the TBUF.
3. Validates the TBUF element so that the MSF transmit state machine knows to send the data to the fabric.

If the request is not an SOP, the transmit data thread finds the current packet data location and offset into the buffer and moves all the data from the DRAM to the TBUF and validates the TBUF element. If the request is not an EOP, it places the buffer address onto a buffer free list after the entire packet has been transmitted.

CONCLUSION

Performance was measured for a variety of packet streams, notably single OC-192 flow, 16 flow, and >16 flow cases for minimum-sized packets, and a distribution of packet sizes for non-minimum-sized packets. The context pipe stages tend to govern performance that is a steady rate of one packet per 55.6 ME cycles.

The elasticity buffers need to accumulate messages before the following context pipes could achieve a full rate without getting empty ring messages.

The 10 Gigabit/sec performance is achievable with careful attention to the instruction counts for context pipe stages, and the overall latency/number of threads for the functional pipe stages.

ACKNOWLEDGMENTS

The authors acknowledge Gil Wolrich, Hugh Wilkinson, Deb Bernstein, Michael Fallon, Sanjeev Jain, Stepanie Hirnak, David Romano, Mark Rosenbluth, and John Wishnewski.

REFERENCES

- [1] Matthew Adiletta, et al., "The Next-Generation Family of Intel Network Processors," *Intel Technology Journal*, Q3, 2002, V6 Issue 3.
- [2] Internet Network Working Group, RFC 2697, September 1999.
- [3] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, V.1 N.4, August 1993, pp. 397-413.
- [4] Internet Network Working Group, RFC 2863, April 1998.

AUTHORS' BIOGRAPHIES

Matthew Adiletta is an Intel Fellow and Director of Communication Processor Architecture. He led the architectural development and implementation of the IXP2800 and is driving the IXP roadmap. He is interested in processor architecture and advanced implementation techniques for rapid silicon development. He is also intrigued with the semantic web and network security. Adiletta received his B.S. degree in Electrical Engineering, with Honors, at the University of Connecticut. He resides in Bolton, Massachusetts. His e-mail is matthew.Adiletta@intel.com.

Donald Hooper is a senior software architect in the Network Processor Group. He has led many projects including Logic Synthesis, Video Servers, MPEG 2 and DAVIC Standards, IXP1200 Software Tools and Libraries, IXP2800 Proof of Concept Designs, and NPG Coding Standards. His professional interests include networking, artificial intelligence, and object-oriented languages. He attended four colleges with cumulative B.S.E.E. degree credits finishing at UCLA. He resides in Shrewsbury, Massachusetts. His e-mail is donald.hooper@intel.com.

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