

Intel[®] Technology Journal

Network Processors

**Network Processor Building Blocks
for All-IP Wireless Networks**

Network Processor Building Blocks for All-IP Wireless Networks

Harsh Vipat, Philip Mathew, Manohar Ruben Castelino, Auro Tripathy
Intel Communications Group, Intel Corporation

Index words: IXP2400, network processor, 3G, RNC, IPv6, microblock

ABSTRACT

The focus of this paper will be on the hardware features of the IXP2400 network processors and how they help to accelerate the key processing needs of wireless networks. The first part of the paper will explain the 3G wireless network topology and the role of network nodes such as base stations, radio network controllers, and routing gateways. The next part of the paper will identify a wide range of packet processing functions performed at the radio network controller (RNC) node. The heart of the article will delve into the inter-workings of seminal reusable network-processor-based building blocks such as packet forwarding at layer-3, bandwidth-saving header compression and decompression schemes, IPv6-to-IPv4 tunneling, and QoS. The article will conclude by asserting that all this, coupled with programmability and hardware acceleration capability, meets the evolutionary needs of wireless networks.

INTRODUCTION

Traditional wireless telecommunication networks and data communication networks will converge. The most recent specification of the converged network (also known as

3GPP Release 5) specifies a wireless network where the transport layer utilizes Internet Protocol (IP) networking as much as possible. In this all-IP network, both user data flows and control flows will be based on IP, thus making the end-to-end network a packet-switched IP network. In practice, the single most important packet data protocol to be supported is IPv6. A simplified reference model for the General Packet Radio Service (GPRS) network is shown in Figure 1.

The role of the mobile terminal in an all-IP network is to originate and terminate both connection-oriented (TCP/IP) and connectionless (UDP/IP) real-time and non-real-time services such as web browsing and VoIP calls.

The rest of the components of a wireless network can be broadly categorized into the Radio Access Network (RAN), comprised of the Base Station and the Radio Network Controller (RNC) and the Packet-Switched Core Network comprised of the Serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN).

Base stations (also known as Node Bs) link the mobile terminal to the rest of the fixed and mobile network. Each base station provides radio coverage to a geographical

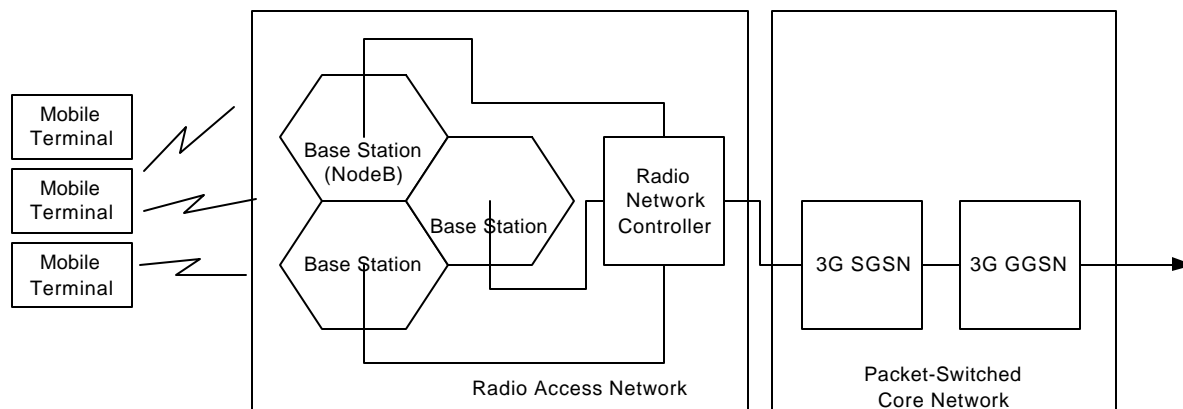


Figure 1: Nodes of 3G wireless network

area known as a cell. A typical network may be visualized as consisting of a mesh of hexagonal cells, each with a base station at its center. Base stations are connected to one another by central switching centers known as Radio Network Controllers (RNCs). The RNC (Figure 2) manages the wireless radio interfaces of the base stations and controls handoff, sending data from the core network to one or more base stations in the forward direction, and selects the best signal from several base stations and sends it to the core network in the reverse direction. For example, if a mobile terminal user moves out of one cell and into another, the RNC hands over communication to the adjacent base station (the switching function). Alternately, the RNC may route calls (packets) to another RNC in the network.

In the core network, the SGSN takes care of routing, handover, and IP address assignment. For example, if you were in a car on the highway and were browsing the Internet on a mobile terminal, you would pass through many different cells. The SGSN routes the packets to the appropriate base station and maintains a seamless connection.

The GGSN is the “port of last call” in the Core Network before a connection to an ISP or corporate network’s

router occurs. The GGSN is basically a gateway, router, and firewall rolled into one.

While the packet processing needs of various nodes in a wireless network are unique, the processing needs of the Radio Network Controller (RNC) are most challenging. The RNC serves as a transition point between the predominantly IPv6 Radio Access Network (RAN) and the Core Network (CN), which has both IPv6 as well as IPv4 traffic. The RNC also handles both packetized voice and data packet flows, which have different requirements in terms of delay and loss characteristics. The processing functions at the RNC therefore include IPv6 routing, IPv4 routing, header compression and decompression, tunneling and QoS. All these functions can be implemented as reusable building blocks on IXP2400.

The rest of the paper is organized as follows. First, we take a brief detour and provide overview of the IXP2400 network processor and a possible software architecture for implementing the building blocks. The remaining sections cover the processing functions of RNC nodes and highlight the specific hardware features of IXP2400 that can be utilized to implement them efficiently.

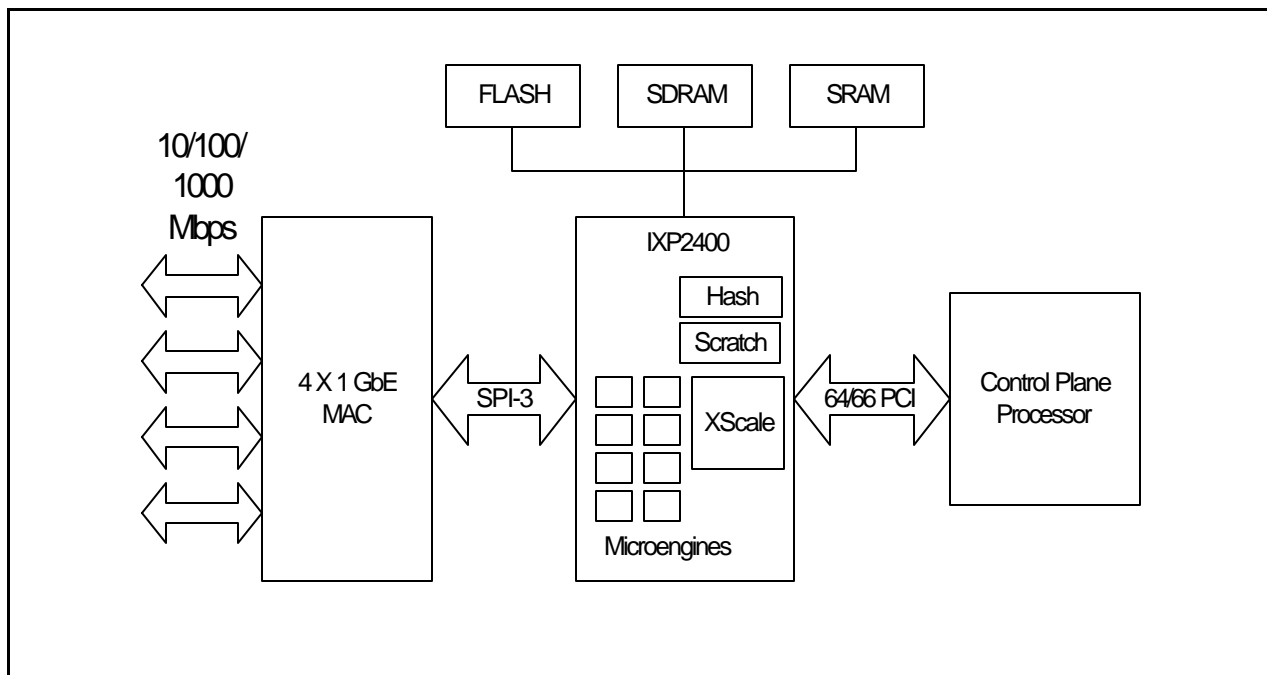


Figure 2: Conceptual block diagram of IXP2400-based RNC

OVERVIEW OF THE IXP2400 PROCESSOR

The IXP2400 has eight programmable second-generation microengines (MEv2) and an integrated XScale™ core. The common processing, called fast-path processing, performed on a majority of the packets, is implemented on microengines. Housekeeping functions and exception processing (performed on a minority of the packets) are typically implemented on the XScale.

Each microengine has eight threads, and each thread has its own hardware context consisting of a register set, program counter, etc. The context swaps are therefore very inexpensive. The threads are scheduled non-preemptively by the hardware. The microengines also have a fully associative 16-entry CAM and 640 32-bit words of local memory to speed up stateful packet processing.

On a network processor, the packets can arrive at a specified maximum line rate. The processor must perform the required processing on these packets and must transmit the processed packets in sequence at the desired rate. When the time required to process each packet far exceeds the inter-packet arrival time, a software pipeline architecture can be deployed to achieve required line rates.

In a software pipeline model, the processing required for a packet is divided into several sequential stages. Each stage provides only a part of the entire processing, and the packets therefore go through all the stages to complete entire processing. There are two basic pipelining approaches: context pipelining and functional pipelining. In a context pipeline, each stage is implemented on a microengine, and each microengine therefore works on different stages of a packet. In functional pipelining, the same microengine processes different stages of a given packet.

Fundamental to implementing software context pipelines across microengines are IXP2400 features such as hardware-assisted scratch-memory resident producer/consumer rings (referred to as scratch rings) and private registers between adjacent microengines (referred to as next-neighbor register rings) that allow efficient implementation of producer and consumer communication. These hardware-managed mechanisms rapidly pass state from one microengine to the next. The IXA SDK provides a framework for implementing processing functions as reusable building blocks (microblocks) and for combining them in desired fashion to form functional pipelines. The following sections describes how the functions of RNC nodes can be implemented using a combination of context and functional pipelining.

PROCESSING REQUIREMENTS OF RNC NODES

The packet processing in an RNC node (Figure 3) can be divided into four major stages: the receive stage (Rx), the header processing stage, the QoS stage, and the transmit stage (Tx). The receive stage is responsible for layer-2 reassembly and framing. The header processing stage can include a variety of functions such as layer-3 forwarding, header compression, etc. The QoS stage provides priority queuing of packets based on classes. The transmit stage is responsible for transmission of the frames. Each of these steps can be implemented as a context pipeline stage on a set of microengines on IXP2400.

The Rx and Tx functions are link-layer specific and can be easily implemented with the help of features provided by the media and switch fabric interface available on IXP2400. The header processing and QoS stages involve more complex and stateful processing and will be the focus of following subsections.

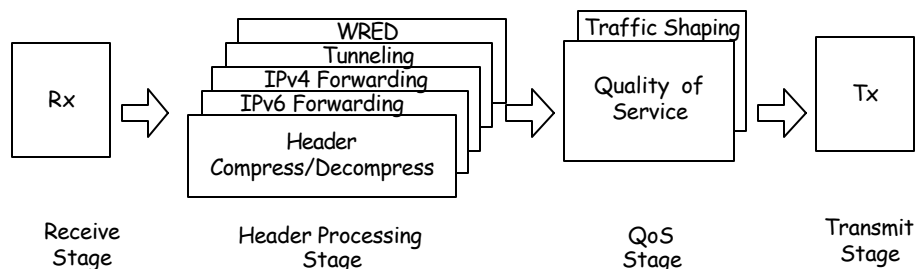


Figure 3: RNC processing stages

Header Processing Stage

Figure 4 shows the block diagram of the header processing stage. The header processing stage can be implemented as a functional pipeline, and each processing function is therefore implemented as a microblock. The following sections describe details of each block and the hardware features that are relevant for the efficient implementation of the block.

Link-Layer Decapsulation and Classification

Since the RNC connects the radio networks to the core network, the packets arriving at the RNC may have different link-layer encapsulations. This poses a challenge for the next stage, namely, header processing. All the processing blocks in the header-processing stage primarily process layer-3 and higher layer headers. Due to different sizes of link headers, the offset of the layer-3

are the byte_align instruction, index mode addressing of registers, and local memory. The byte_align instruction allows concatenation of data in two 32-bit registers and extraction of any four bytes from a concatenated string into the destination register. The byte index for alignment can be selected by setting a control status register (CSR). The IXP2400 also allows indirect referencing of the transfer registers. An index pointer can be set to point to a particular transfer register by writing to a control status register. The transfer register can then be accessed by indirect referencing through the index pointer. Auto increments and decrements of the index pointer are also supported.

The local memory is addressable storage located in the microengine. The local memory can be accessed at long-word (32-bit) granularity, and the latency cost of an access is the same as accessing general-purpose registers. Two

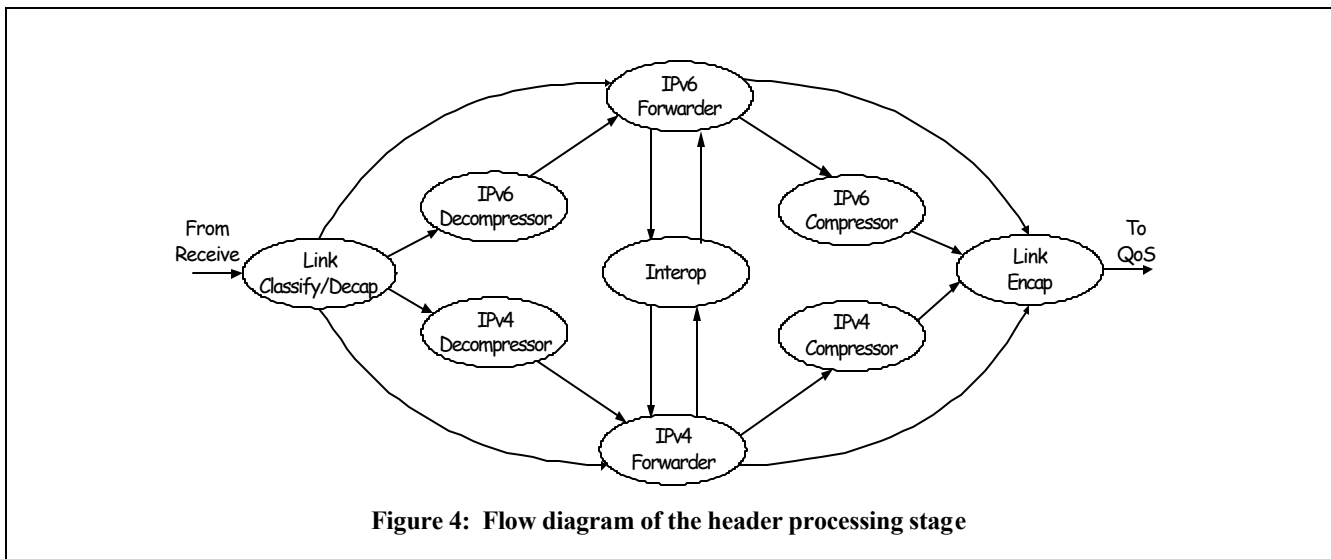


Figure 4: Flow diagram of the header processing stage

header can only be determined at run-time. This unnecessarily requires each processing block to know about different link encapsulations and deal with unaligned headers. The link classify block solves this problem by aligning and caching the layer-3 and higher headers in the local memory of the microengine. This is advantageous in many ways. The processing blocks can access the header fields with a minimum of one cycle latency. The headers can be treated as global data structures and can be shared by all the processing blocks in the pipeline without the copying and aligning overhead that each block downstream must undertake. Lastly, this approach also makes the processing blocks link-layer independent and enhances their reuse potential.

The key architectural features that allow efficient implementation of the aligning and caching functionality

index registers are available to the programmer, which can be set to point to any of the 640 long-words of local memory. The local memory can be accessed by dereferencing the index register, or by specifying offset using array notation.

IPv4 and IPv6 Layer-3 Forwarding

The RNC has to route packets between the predominantly IPv6 RAN and the Core Network (CN), which has both IPv6 as well as IPv4 nodes. The IPv6 and IPv4 forwarding are therefore the key functions of the header-processing stage. The aggregation of IP address space requires use of a special search technique called Longest Prefix Match (LPM). Routing prefixes (route entries) are stored in a route table along with their associated next-hop forwarding information. The route table is searched to find

a longest prefix that matches with the destination address. The next hop information associated with the matched routing prefix is used to forward the packet. In order to facilitate LPM, routing prefixes are usually stored in complex tree-like data structures in SRAM. As a result, SRAM is accessed several times during the LPM.

Hardware-assisted multi-threading available on IXP2400 processors allows efficient implementation of memory-intensive algorithms such as LPM. A thread-issuing memory operation can explicitly yield control to other thread and allow some other processing to take place while the memory operation completes. The effective latency cost of the memory operations can thus be reduced or even be eliminated with the help of hardware-assisted multi-threading. Large numbers of GPRs allow efficient handling of long 128-bit IPv6 addresses. Since IPv6 supports hierarchical addressing and address aggregation in a structured way, the local memory can play an important role in optimizing the IPv6 LPM. Depending on the position of the router in the address hierarchy, the aggregation information can be cached in the local memory, and the route lookup can be speeded up substantially.

The layer-3 forwarders also have to handle a variety of exception conditions. These include processing of IP options, dealing with fragmentation and reassembly, processing dynamic route update requests, and responding to Address Resolution Protocol (ARP) or neighbor discovery messages. All these functions can be implemented on the XScale™ core. Once again, hardware-assisted scratch rings help in implementing communication between microengines doing the fast-path processing and the XScale core doing the exception path processing.

Header Compression and Decompression

Two factors contribute to the use of header compression schemes in wireless network. The first factor is that, for the IPv6 packet, the IPv6/UDP/RTP header is 60 bytes in size and a typical speech payload is about 20 bytes in size, a 300% overhead! The second factor is that wireless spectrum is expensive. Header compression schemes reduce the header to three bytes, yielding a manageable overhead of 15%.

Header compression schemes are based on the observation that many fields of the protocol headers rarely change during the life of a session. Also, many other fields change only in small, predictable quantities.

Compression and decompression are enabled by creating and storing a compression context for the RTP session at the compressor and the decompressor. A compression context has two parts: the context identifier and the

context information. The context identifier, a unique number denoting an RTP session, is derived from several fields of the header. For example, for RTP-based voice packets, the context identifier is derived from the source IP address, destination IP address, source port, destination port, and Synchronization Source Identifier (SSRC) fields. The context information includes all the fields in the IP, UDP, and RTP header. The very first packet of a session transmits the context identifier embedded in the uncompressed packet. Once the compressor and the decompressor get the context information associated with that context, compressed packets carry only the context identifier (pertaining to that RTP session) and the differences of the changing fields. The decompressor uses the context identifier to regenerate the full header.

In the IXP2400, multiple threads perform packet compression or decompression in parallel. The challenge is to ensure that the packets are exiting the compressor or the decompressor in the same order in which they entered. For every packet, threads need atomic access to the context information tables. This involves multiple accesses to high-latency external memory such as SRAM or SDRAM. Yet another factor slowing down performance is ensuring that packets are received by the threads in order, i.e., the first thread receives the packets, completes the critical section, and then signals the second thread, and so on. A method called *folding* described in the next section addresses this.

Folding

The principle of folding (or memory coalescing) provides that if a thread has already requested access to an external memory location (to be fetched into local memory), then other threads requesting access to that same memory location can simply wait to access it in local memory. In the meantime, they can yield to other threads. A combination of the microengine Content Addressable Memory unit (CAM unit) and local memory unit can be used to implement folding. A lookup into the 16-entry CAM results in a CAM miss or a CAM hit coupled with an index into local memory where the data resides. Thus, the microengine local memory acts as a cache, and the CAM aids in the cache lookup for every packet before accessing the external memory unit.

Folding can reduce accesses to external memory. The context identifier (uniquely identifying the context information of an RTP session) resides in the CAM and is used as a tag to the context information residing in local memory. The CAM holds the local memory index and a reference count. Processing is divided into two phases: a read or populate phase and a consume phase. In the read or populate phase, each thread of a microengine looks up the context identifier in the CAM and, in the case of a

miss, adds the identifier to the CAM. In the case of a miss, it also issues a read to the external memory unit for loading the context information associated with the context identifier to the local memory. In the case of a CAM hit, the thread simply increments the reference count of the CAM indicating its interest in the same data.

In the consume phase, threads access the data already available in the local memory and decrement the reference count. If the reference count of the CAM entry is zero, it is the last thread accessing this data; hence, it flushes this data to the SRAM unit. The benefit of this scheme is that

multiple packets refer to the same context information in local memory (instead of SRAM). Data is read only once from the external memory unit into the local memory. All subsequent modification of the data occurs in the local memory. Finally, one or more writes from local memory to external memory are performed depending on the CAM eviction policy. Figure 5 illustrates this approach in detail.

The hash engine or the CRC engine available in the IXP2400 can be used to generate the 32-bit unique context identifier given a 5-tuple from the RTP header.

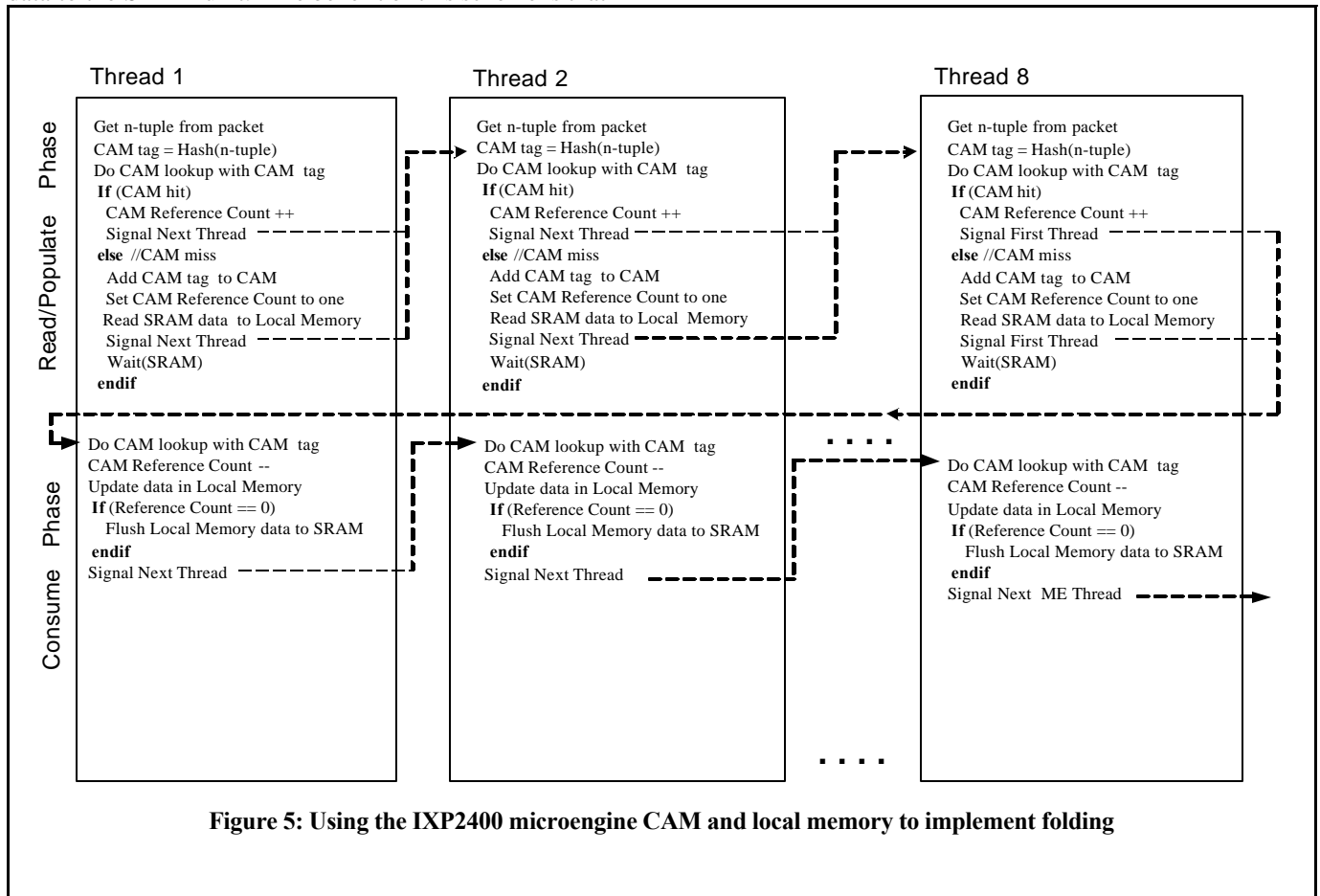


Figure 5: Using the IXP2400 microengine CAM and local memory to implement folding

Another technique to speed up processing is to cache the compressed/decompressed packet headers in the microengine local memory so that the compressor/decompressor need not access the packet header from the external memory unit.

IPv6 over IPv4 Tunneling (Interoperability)

Driven by a need for large numbers of uniquely addressable wireless devices, IP networks will gradually transition from a pure IPv4-based network to an IPv6-based network. The key to a successful transition is

interoperability with the large installed base of IPv4 hosts and routers. Maintaining interoperability with IPv4 while deploying IPv6 will streamline the task of transitioning the Internet to IPv6. This can be achieved using multiple mechanisms. The IPv6-IPv4 interoperability block (see Figure 4) uses IPv6 over IPv4 tunneling (encapsulating IPv6 packets within IPv4 headers to carry them over IPv4 routing infrastructure) to help realize this transition.

The main function of this block is to provide encapsulation and decapsulation of IPv6 datagrams. The

functionality includes removing and attaching appropriate IPv4 headers to IPv6 datagrams. Similar to link-classify and encapsulate blocks, the challenge is to be able to read or write layer-3 headers at arbitrary byte offsets. Once again, byte_align instruction, index mode addressing, and caching of protocol headers in local memory allow efficient implementation of these blocks.

This block integrates seamlessly with the IPv4 and IPv6 forwarding blocks. The IPv6 over IPv4 tunnels are configured by using special IPv6 and IPv4 route table entries. Hence the IPv6 and IPv4 forwarders are used to direct traffic transitioning from IPv6 to IPv4 networks and vice-versa to the interop block (see Figure 4). The IPv6 and IPv4 forwarders can also be configured to run independently (dual IP layer) on the same router. This dual IP layer functionality is also a requirement for successful transition from an IPv4 to IPv6 Internet.

Additional transition blocks can leverage the existing blocks to support advanced mechanisms like Network Address Translation-Protocol Translation (NAT-PT). Also, as the transition mechanisms are evolving, the programmable IXP2400 makes the task of adding software blocks supporting newer and more efficient transition mechanisms relatively painless. The possibility of being able to reprogram the IXP2400 in the field to support evolving networking standards is key to achieving rapid transition to the new standards.

THE QoS PROCESSING STAGE

The 3G wireless network defines four QoS classes: a low-delay conversational class for voice traffic, a constant delay streaming class for streaming video, a payload-preserving interactive class for web browsing, and a best-effort background class for e-mails and downloads. While the IXP2400 is fully programmable, the QoS software building blocks can be designed to be configurable to meet the needs of the 3G QoS classes. This section describes QoS building blocks on the IXP2400 and how they can be configured for 3G QoS.

The three functional blocks associated with QoS processing are the queue manager, the scheduler, and the rate shaper (Figure 6).

The queue manager block performs the packet enqueue and dequeue operations and updates and maintains queues. Fortunately, the SRAM Q-Array hardware-assist can be used for this purpose. The SRAM Q-Array hardware is used to cache the most recently used 64 queues in the SRAM controller. The Q-array data structure caches the head and tail pointers of the queue, as well as the number of entries currently present in that queue. Caching queue entries (in the memory controller) helps components like WRED and QoS, which manipulate large number of queues. The Q-Array hardware along with the CAM (local to each microengine) enables them to cache and access these queues efficiently.

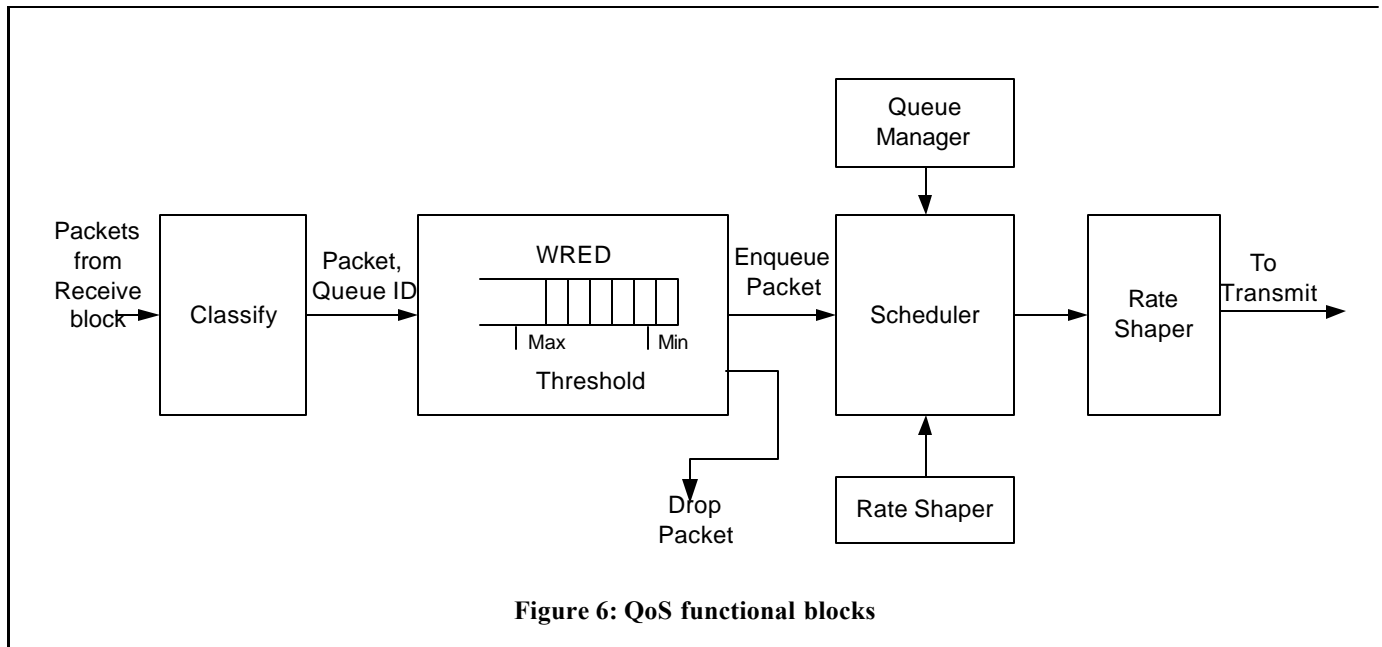


Figure 6: QoS functional blocks

The queue manager receives requests to enqueue packets from the packet processing threads through the hardware-assisted producer consumer rings. Requests contain packet handle and queue number, to which packets must be enqueued. The queue manager also processes requests from the scheduler to dequeue packets.

In addition, the queue manager maintains the queue transition information that will be used by the scheduler. A queue transition occurs either when a packet is enqueued to an empty queue or when the queue is emptied out by dequeue of the last packet in a queue. Furthermore, the queue manager uses payload size information returned from dequeue operations to update credit information used for scheduling decisions and to update rate-shaping information.

The packet scheduler can be configured for Deficit Round Robin (DRR), Round Robin (RR), multi-level hierarchy (as in DiffServ), or any other proprietary scheduling scheme. Because the scheduler cannot tolerate any inherent delays, all its internal data structures such as queue status vectors, credit status vectors, shaping status vectors, and Round Robin masks are maintained in single-cycle access local memory.

The rate shaping is used to limit the rate at which packets are sent out on a virtual interface. This is needed when a single physical link aggregates one or more virtual links that are demultiplexed at the other end of the link. When a packet is scheduled to transmit on an interface, the shaper calculates the time slot at which packet transmission can be allowed the next time and registers that interface in the timing calendar queue. The rate shaper also turns the interface off to stop any further scheduling on this interface. When the timer reaches the appropriate time-slot entry in the timer calendar, it turns on the interface again. The microengine architecture provides a hardware timer capable of signalling at a period of 16 clock cycles. The current time (in ME cycles) is accessible in 3 cycles and can be used to accurately control the rate at which packets are sent out.

In addition, congestion avoidance mechanisms such as Weighted Random Early Detection (WRED) can also be implemented in the packet processing stage. The IXP2400 microengines use the well-known technique of using the Linear Feedback Shift Register (LFSR) hardware to generate extremely good 32-bit pseudo-random patterns. This hardware supplies the random-number to the microengines in two clock cycles. The pseudo random number generator is used in components such as WRED, which require an efficient random number generator to compute packet drop probability.

Each queue or interface on the system can be configured to have a different set of WRED thresholds, drop probabilities, and scheduler priorities. The rate shaper can also be configured to support a different transmit rate on each logical interface.

This ability to configure WRED, scheduler, and rate shaper allows the implementation of the 3G QoS classes. For instance, the low-delay conversational class traffic can be classified and placed into queues that have lower WRED thresholds and higher scheduler priorities. The lower WRED thresholds ensure lower latencies. The higher priority assigned to these queues ensures that the packets are transmitted ahead of others. On the other hand, best-effort traffic can be classified into queues with high WRED thresholds and placed in the best effort queue.

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

CONCLUSION

The IXP2400, a programmable network processor with a comprehensive set of hardware assists (summarized below) is ideal for present as well as future needs of evolving all-IP 3G wireless networks. Even after implementing the key processing functions, sufficient headroom remains available for adding new services such as Multi Protocol Label Switching (MPLS), and the upgrades can be performed in the field. This allows nodes such as RNCs to grow and evolve with the upcoming roll out of wireless services.

Table 1 summarizes packet processing functions and the IXP2400 hardware feature that can be utilized to implement them efficiently.

Table 1: Packet processing functions

Processing Block	IXP2400 Hardware Assists
Link-layer classify and encapsulation/decapsulation	Hardware-assisted scratch rings to dequeue packets from Rx. Single-cycle byte-align instructions and index-mode addressing to read layer-3 headers at arbitrary offset. Local memory to cache headers so that other blocks can use them.
Layer-3 forwarding	Hardware multi-threading to hide memory latencies. Caching packet headers and

Processing Block	IXP2400 Hardware Assists
	route-table data in local memory. Hardware-assisted scratch rings for sending exception packets to and from ME to XScale™.
Header compression/ decompression	CAM and local memory for caching. Hardware-assisted hashing for generating compression contexts.
QoS	SRAM Q-array for hardware-assisted enqueue and dequeue. CAM and local memory for caching the most active queues. Fine granularity timer to shape traffic. Local memory for real-time access to scheduling and shaping data structures.
WRED	Pseudo-random number generator to calculate drop probability.
IPv6 over IPv4 Tunneling	Local memory to cache layer-3 headers. Single-cycle byte-align instructions to assist encapsulation and decapsulation of IPv6 frames in IPv4 frames at arbitrary offsets. XScale to dynamically configure tunnels and routes.

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

ACKNOWLEDGMENTS

The authors acknowledge the contributions of Uday Naik, Alok Kumar, Chen-Chi Kuo, Larry Huston, Philip J. Young, Sridhar Lakshmanamurthy, and Makaram Raghunandan.

REFERENCES

- [1] Third-Generation Partnership Project, Technical Specification Group Services and System Aspects, QoS Concept and Architecture (Release 5), 3GPP TS 23.105 V5.1.0.

- [2] Third-Generation Partnership Project, Technical Specification Group Radio Access network, IP Transport in UTRAN (Release 5), 3GPP TR 25.933 V5.0.0.

- [3] “Compressing IP/UDP/RTP Headers for Low-Speed Serial Links,” *IETF RFC 2508*.

- [4] Sally Floyd and Van Jacobson, “Random Early Detection Gateways for Congestion Avoidance,” *IEEE/ACM Transaction on Networking*, August 1993.

AUTHORS’ BIOGRAPHIES

Harsh Vipat is an applications engineer in the Network Processor Group. His interests include networking, operating systems and distributed systems. He has a Master’s degree in Computer Science from Arizona State University. He can be reached via e-mail at harshawardhan.vipat@intel.com.

Manohar Ruben Castelino is an applications engineer in the Network Processor Group. He has worked in projects primarily in the networking and network management areas. His interests include networking and compiler design. He has a B. E. degree from KREC India. He can be reached at manohar.castelino@intel.com.

Philip Mathew is a network application software engineer at Intel’s Network Processor Division. His professional interests include computer networking, embedded programming and object-oriented programming. He received his Master’s degree in Computer Applications from the University of Calicut, India, in 1994. He can be reached via e-mail at philip.mathew@intel.com.

Auro Tripathy is an applications engineering manager in the Network Processor Group. He has led projects in the wireless and broadband access aggregation areas. His interests include networking, digital video, and embedded developments tools and languages. He has a B. Tech degree from IIT Kharagpur, India, and an MSCS degree from Wayne State University, Detroit. He resides in Milpitas, California, and can be reached via e-mail at auro.tripathy@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