

Intel[®] Technology Journal

Network Processors

IXA Portability Framework: Preserving Software Investment in Network Processor Applications

IXA Portability Framework: Preserving Software Investment in Network Processor Applications

Uday Naik, Alex Shoykhet, Larry Huston, Donald Hooper, Raj Yavatkar, Duke Tallam,
Travis Schluessler, Prashant Chandra, Adrian Georgescu
Intel Communications Group, Intel Corporation

Index words: software framework, network processor, data plane, microblocks

ABSTRACT

Network processors are being targeted at a wide range of applications with varying packet processing and throughput requirements. The programmability and flexibility of a network processor make it suitable for applications ranging from Voice over IP to mobile IPv6 with data rates spanning OC-3 to OC-192. In such an environment, the investment made in software development by equipment manufacturers is increasingly significant. Preserving this investment and leveraging it across multiple projects are key considerations when making the right choice of a network processor.

The IXP family of processors provides a very powerful and scalable solution for the network processor market. The IXP2400 targets data rates from OC-3 to OC-48, while the IXP2800 is designed for applications with data rates ranging from OC-48 to OC-192. The IXA portability framework provides the associated software infrastructure to help develop modular, reusable software building blocks for these processors.

This paper describes how the IXA portability framework helps accelerate software development and improves code reuse across applications. The paper details how applications written to the IXA portability framework may be scaled up or down to fit the needs of a specific application.

The paper describes a reference application—IP forwarding with DiffServ—and its implementation in a number of different configurations including dual-chip IXP2400 at OC-48 rates, single-chip IXP2400 at 2 x OC-12 rates and single-chip IXP2800 at 2 x OC-48 rates.

The paper illustrates how the design of the application may be structured to facilitate reuse of software blocks

across these configurations. Using this design as an example, the paper describes various features of the IXA framework and its value in the software development cycle.

The paper assumes that the reader is familiar with the Intel Network Processor Family [1].

INTRODUCTION

A networking application typically operates on three logical planes (Figure 1):

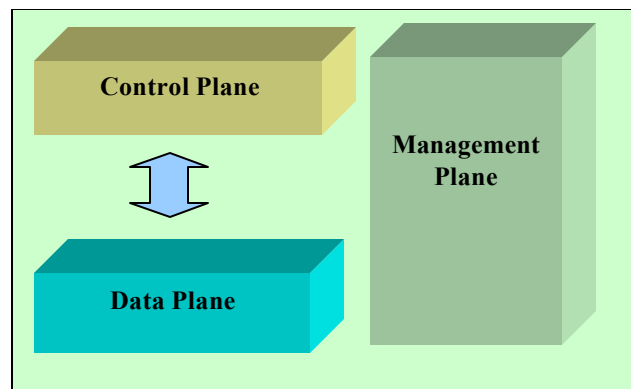


Figure 1: Logical planes in a networking application

1. The *data plane* processes and forwards packets at high speed. It is typically the most performance sensitive since all packets processed by the device must pass through here. In the IXP family, the data plane consists of

the *fast path*, i.e., the MEv2 microengines, which handle most of the packets. For example, the fast path handles the simple forwarding of IPv4 packets.

the *slow path*, i.e., the XScale™ core. This handles a few packets that cannot be handled on the fast path because of the complexity of the processing involved.

These packets are called *exception packets*. Examples include forwarding IP packets with options in the header, packets that need to be fragmented, etc.

2. The *control plane* handles protocol messages and is responsible for the setup, configuration, and update of tables and data sets used by the data plane for lookups. For example, the control plane processes Routing Information Protocol (RIP) packets and updates the IPv4 forwarding table used by the data plane. In the IXP environment, the XScale core may function as the control plane, or this functionality could be supported by an external processor.

3. The *management plane* is responsible for system configuration, gathering and reporting statistics, and stopping or starting applications in response to user input or messages from other applications.

The IXA portability framework is used primarily to develop data plane software and to help interface with code running on the control plane.

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

FAST PATH: MICROENGINE FRAMEWORK

Developers writing microengine code must consider the tradeoffs between modular, easy-to-maintain, and reusable code versus performance. For bleeding edge applications, it may be important to get the highest level of performance possible. For applications in which new features will be added over time, modularity and code maintenance are likely to be more important.

The IXA portability framework is intended to help address both these needs for customers by offering a tiered solution where the programmer can use as much of the framework as appropriate for the application. As more of the framework is included, the developer will get more modular and reusable code.

The IXA portability framework consists of different layers that the developer can choose to incorporate into the design. One layer consists of the tools including support for a high-level programming language. The next layer is a set of libraries that are optimized for the particular network processor. The last layer is a programming model called microblocks.

We describe each of these layers in the following sections.

High-Level Language Support: Microengine C Compiler

At the lowest level of the framework are the tools. The IXA portability framework provides both an assembler and

a C compiler for the microengines. By providing a high-level language through C, the framework helps to abstract some of the underlying hardware details from the programmer. There are significant advantages to using C over microengine assembly.

C is the programming language of choice for most embedded system and network application developers. Availability of a C compiler reduces the learning curve for programmers new to the IXP environment.

A high-level language is much more effective at abstracting and hiding the details of the microengine instruction set from programmers.

It is typically easier and faster to write modular code and maintain it in a high-level language with its superior support for data types, type checking, etc.

However, since code running on the microengines is on the packet processing critical path, it is extremely important to generate very efficient code that meets the performance requirements of the application.

Intel's Microengine C Compiler is specially optimized for the IXP2400 and IXP2800 microengine environment. For example:

The compiler allows the programmer to specify which variables must be stored in registers and which may be stored in memory.

The compiler allows the programmer to specify the type of memory (off-chip SRAM, DRAM, or on-chip scratch memory) that should be used to allocate a specific variable or data structure. Each type of memory has different latency and access characteristics, and the compiler gives developers more control in laying out the data structures across different types of memory.

To handle hardware specific features, the compiler supports intrinsics and inline assembly. The use of these, however, undermines the portability of the code to future generations of network processors. In this case, the data plane libraries provide the only isolation from the hardware and are critical for code portability. For this reason, the IXA portability framework supports the data plane libraries in both microengine assembly and microengine C.

The compiler has a packet format for bitfield structures. Unlike standard C bitfield structures, where the fields must line up with a 32-bit boundary, the fields of a packed structure in microengine C have no such restriction. This is highly suitable for defining and accessing fields of protocol headers.

In summary, the Microengine C Compiler allows developers to write code in a high-level language while still meeting the performance expectations of fast-path packet processing.

Low-Level Support APIs: Data Plane Libraries

The next component of the IXA portability framework consists of the data plane libraries. These are libraries that abstract out some of the implementation details of the hardware, as well as provide common building blocks that many applications will need. These libraries are provided as assembler macros and C functions.

The data plane libraries are optimized for maximum performance and minimal code space utilization. They are the foundation for all MEv2 microengine code. The libraries consist of several portions that enhance code portability and reuse (Figure 2).

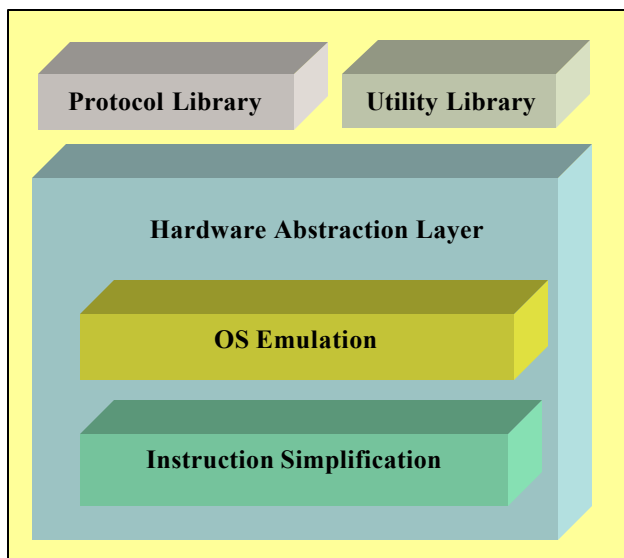


Figure 2: Low-level libraries on the microengine

The instruction simplification library provides assembler macros that simplify common microengine operations such as loading constants, accessing Chip-Specific Control and Status Registers (CSRs), and performing indirect operations. This library isolates developer and the software they write from changes to the instruction set. The use of the data plane library ensures that the code is not only portable to the new processor but also able to perform better with the latest instruction set.

The Operating System (OS) emulation library contains support for services that would normally be provided by an OS. This library uses the underlying hardware to support inter-thread signaling, messaging, synchronization (locking and critical sections), queuing, buffer management, timers, etc. As with instruction

simplification, the use of the library isolates the programmer from changes to future hardware implementations.

The utility library includes support for hash table lookups, endian swaps, CRC (Cyclic Redundancy Check), and other useful functions.

The protocol library provides an optimized implementation of many common networking functions. Some examples of these are functions that parse and modify IP packet headers.

Modular Building Blocks: Microblocks

The highest layer of the IXA portability framework for the microengines is the programming model and associated support. In this programming model, the developer divides the fast-path processing of the application into high-level logical components called *microblocks*.

The programming model enables and requires microblocks to be written such that each microblock is independent of others. By providing clean boundaries between these blocks, the programming model makes it possible to modify, add, or remove more microblocks without affecting the behavior of the other blocks. This improves reusability and allows developers to combine microblocks and create different applications. The net benefit is that the task of writing fast-path code is simplified and time-to-market is accelerated.

Each microblock is an assembler macro or C function written using the underlying low-level data plane libraries. Note that as opposed to a low-level function, e.g., `ipv4_checksum()`, a microblock is coarse-grained and has state and associated data structures typically shared with the XScale™ core. Examples of microblocks include IPv4 forwarding, Ethernet layer-2 filtering, 5-tuple classification, Multiprotocol Label Switching (MPLS) label insertion, etc.

A microblock has an associated management component on the XScale core. The application is typically written so that the microblock will process most common cases and pass exception cases to the XScale component for further processing.

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

DESIGNING FOR MODULARITY AND REUSE

Figure 3 shows an IP DiffServ application implemented using microblocks. We will use this application to illustrate some important design considerations for modular code.

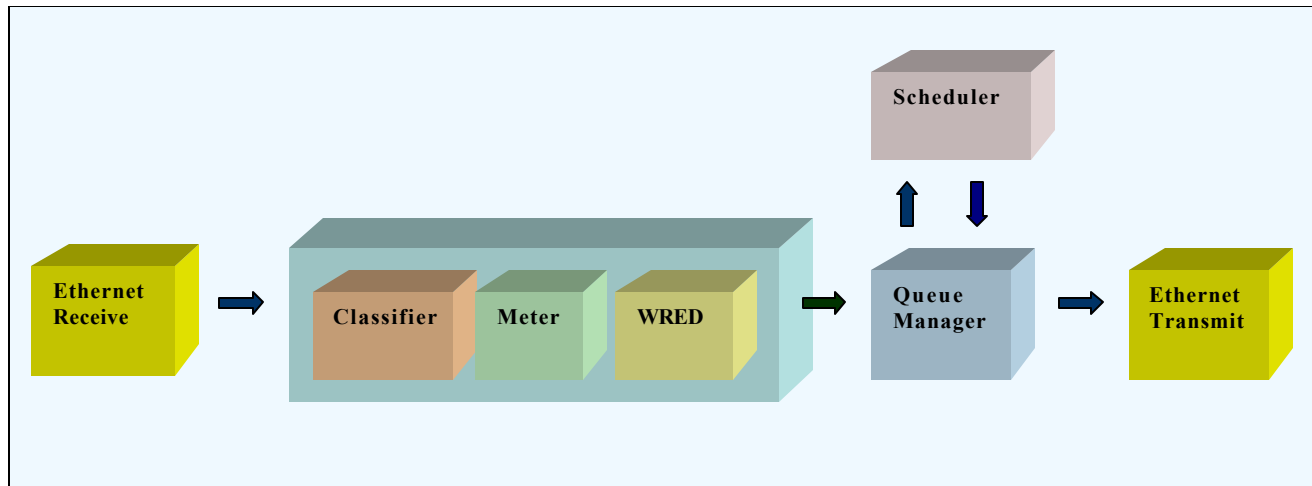


Figure 3: Single-chip DiffServ application using microblocks

Separate Application-Specific Code from Reusable Code

Applications written to the IXA framework include modular reusable microblocks and glue code that combines these blocks and implements the data flow between them. This glue code is referred to as the *dispatch loop*. The dispatch loop combines microblocks running on the same microengine thread and implements the packet data flow between them.

The same dispatch loop may be instantiated on more than one microengine thread to process multiple packets at the same time. For example, in Figure 3, the classifier, meter and Weighted Random Early Detection (WRED) blocks are combined by a dispatch loop and run on several microengine threads in parallel.

In the IXA framework, application-specific code is implemented in the dispatch loop, while care is taken to preserve the reusability of microblocks across applications.

Support for Shared State between Microblocks

An important aspect of the framework is the mechanism by which state is shared between microblocks. This includes per-packet state, referred to in IXA terminology as *packet meta-data* (packet size), offset to the packet data in memory, destination port, etc. Another example of shared state is packet headers that are read and modified by different microblocks in the pipeline. In the IXA framework, the dispatch loop caches packet meta-data and other shared state in registers and provides access to them through an API. The dispatch loop also supports caching packet headers in local memory or registers. Across microengines, the cached information is flushed to memory or passed along in the message queue between

the microengines. The dispatch loop populates the cache by reading information either from memory or from the message queue. This isolates the microblocks from the layout of the meta-data in memory and from the format of the messages on the message queue between microengines. In addition, the packet header cache and meta-data cache considerably improve performance.

For example, in the DiffServ application in Figure 3, the dispatch loop reads in the IP header from memory and caches it in local memory. This header is then used by the classifier to do a lookup and determine the flow and class information for the packet. The same header is also used by the meter stage, which modifies the TOS/DSCP field in the IP header and recalculates the checksum. The modified packet header is written back by the dispatch loop after the WRED microblock. The dispatch loop caches the packet header and per packet information such as flow and class id in registers. The microblocks (classifier, meter, WRED) simply access these through well-defined API's exported by the dispatch loop.

Decouple Application Data Flow from Microblock Design

Typically, the classification and processing performed in a microblock determines to which microblock the packet goes next. For example, based on the processing in a filter block, the packet may either be sent to an IPv4 forwarding microblock or be dropped.

Care must be taken to decouple this data flow from the design of a specific microblock. To preserve its reusability across applications, a microblock must not require a specific microblock to be run before or after this block. In the IXA framework, the dispatch loop implements the data flow between microblocks. Each microblock indicates only the logical result of its classification.

For example, the filter block indicates the next block-PASS or DENY using a dispatch loop state variable. Based on this variable, the dispatch loop may decide that if the next block is PASS, the packet is sent to the IPv4 forwarding microblock, and if it is DENY, the packet is dropped. Alternatively, the dispatch loop may be modified such that instead of dropping packets that are denied, these packets are sent to the XScale™ core.

Separating Packet-Processing Blocks from Network Interface-Specific Blocks

In any networking application, we can clearly differentiate between blocks that are hardware or network interface specific and blocks that process packets.

Network interface-specific blocks include the receive and transmit blocks for the different media interfaces (POS, ATM, Ethernet, etc). Other hardware-specific blocks include the queue manager block that handles the queuing hardware on the IXP2400/IXP2800. Packet-processing blocks are protocol specific, e.g., IPv4/v6 forwarding, Network Address Translation (NAT), layer-2 bridging, etc.

Network interface-specific blocks (receive and transmit) are typically involved in segmentation and reassembly of a

packet. It is a good design practice to decouple the packet-processing microblocks from the segmentation/reassembly of packets by placing them on separate microengines.

For example, in the IP DiffServ application shown in Figure 3, the classifier, meter and WRED blocks are placed on separate microengines from the receive block and interface to it using a message queue.

This has a couple of advantages:

It builds some elasticity into the packet-processing pipeline and allows sudden bursts of received data to be absorbed. For example, typically, the packet pipeline is expected to sustain an average rate lower than the peak arrival rate of traffic. With this design, only the receive/transmit blocks need to be able to sustain the peak rate. As long as the elasticity provided by the message queue between microengines is sufficient, the packet-processing blocks can run at a lower rate.

The packet-processing blocks may be modified or replaced easily without affecting the rest of the pipeline. Figure 4 shows how the packet-processing blocks in the DiffServ application shown in Figure 3 may be replaced by an IPv4 forwarding block, while reusing the rest of the blocks in the pipeline.

The network interface blocks may change in future revisions of the processor. For example, future revisions of the processor may implement the reassembly in hardware. This design hides these hardware changes from the packet-processing code.

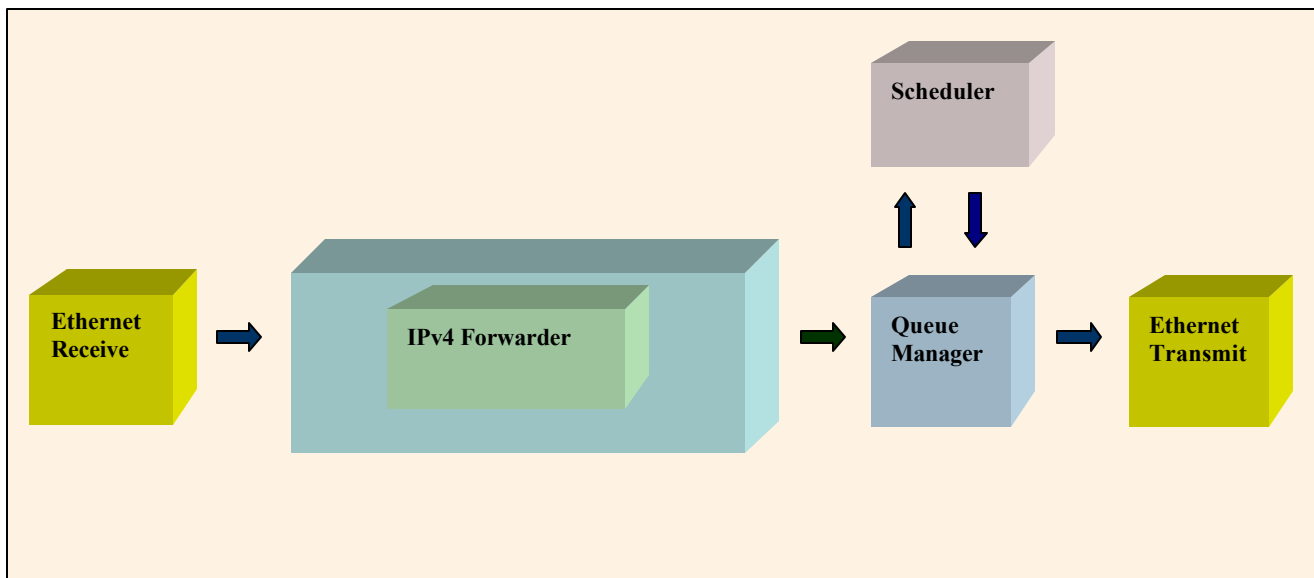


Figure 4: IPv4 forwarding application using microblocks

The network interface blocks may be different depending on the media and bus configurations. These blocks are also slightly different between IXP2400 and IXP2800. The packet-processing blocks, on the other hand, remain the same in all configurations. By separating the two, the design allows for maximum reuse of code.

Since the receive/transmit blocks may need to sustain bursts of traffic, they need to be highly optimized. For this reason, they may be implemented entirely in assembly. By moving the packet processing code to a different microengine, we eliminate the need to mix assembly and C code while combining blocks.

While this approach has the many advantages indicated above, it also implies an extra read and write of the packet header when a minimum size packet is received. For this reason, applications that target full-line rate at OC-192 data rates for minimum size packets may choose to combine the packet-processing code with the packet reassembly [2].

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

SLOW PATH: XSCALE™ CORE FRAMEWORK

In the IXP environment, the XScale may be used for management, control, and even packet processing (Figure 5).

A *core component* is the slow path or XScale counterpart of the microblock and is responsible for its configuration and management. In addition, core components may handle packets that cannot be processed on the microengines because of the complexity involved.

The IXA portability framework provides the support infrastructure and APIs required to develop core components.

Network application developers are commonly faced with the challenge and opportunity to leverage existing code for the slow path and interface it with code written to the IXA portability framework. For this reason, the services provided by the framework on the XScale are divided into discrete layers of functionality. At the lowest layer, the IXA core framework supports a set of APIs called the *resource manager APIs*. These APIs provide support for hardware initialization, configuration, and resource management. They also support communication between the microengines and code running on the XScale. The resource manager APIs isolate the XScale developer from

the specifics of hardware and from the details of microengine to XScale communication.

While a slow path application may be written entirely to the resource manager APIs, the IXA framework also defines a standard modular way of writing XScale core components. The *core component infrastructure library* defines the structure and canonical design of an XScale core component and the mechanism by which messages and packets are passed between core components. Architecturally, a clean interface separates a core component from the rest of the system.

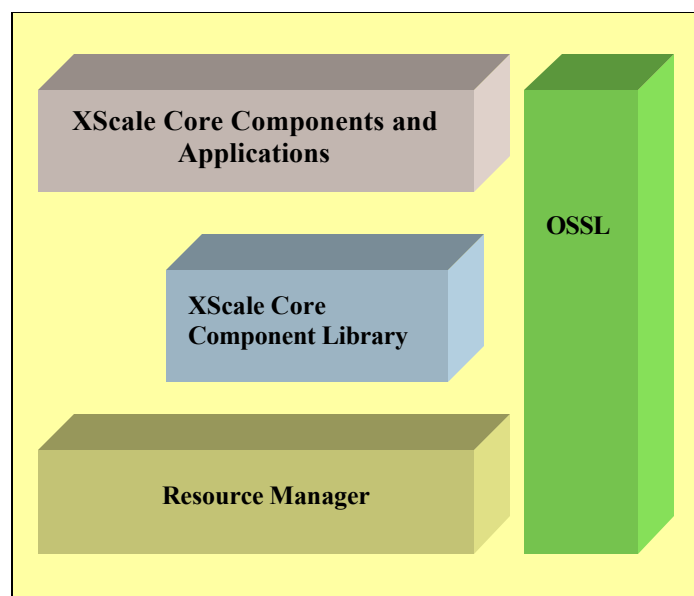


Figure 5: XScale core framework

In most cases, a core component manages a single microblock. The core component/microblock pair may be viewed by the control plane and the rest of the data plane as a single unit of packet processing and reused in a variety of processing configurations. Viewing the pair as a single unit allows an intelligent and highly flexible split of packet processing between the core component and the microblock, depending on the needs of specific applications.

Another important aspect of code reusability on the XScale core is abstraction from the operating system (OS) involved. Depending on the application, the same network vendor may choose a small footprint proprietary microkernel for one product and a much more mature industry standard Real-Time OS (RTOS) for another.

For this reason, the IXA Framework defines an OS Services Layer (OSSL) that provides APIs for commonly

used OS services (e.g., timers, threads, semaphores, etc). Consistent use of this API greatly enhances the portability of the framework (and code written to the framework) to different operating systems.

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

INTERFACING WITH THE CONTROL PLANE

Data plane processing is controlled and managed by control plane software and stacks. In most legacy systems, these typically reside on a separate processor and exchange protocol packets and control messages with the data plane over a control bus or a backplane.

On the IXP family of network processors, some of that functionality may be moved to the XScale™ core. It is important, however, that the control plane is logically separated from the data plane. If the two planes are separated by a set of standard APIs and protocols, equipment manufacturers can choose and upgrade control stacks independent of data plane software. These stacks may come from any vendor who supports these APIs. Upgrades of network processors and migration to newer generations will not require any changes to control plane software.

The Network Processor Forum (NPF), of which Intel is a prominent member, is actively working on defining such standards and APIs. The NPF API that is emerging from this body abstracts the data plane as seen by the control plane and defines per-protocol management interfaces between the two planes.

A working group within the Internet Engineering Task Force (IETF) called Forwarding Control Element Separation (ForCES) is in the process of defining a messaging protocol for control plane or data plane communication. The interconnect-independent nature of this protocol makes it easy to change the bus or the backplane without affecting the software layers that are above this protocol. In fact, the control plane and the data plane software may become located on the same network processor with no changes required, other than to the implementation of the ForCES protocol.

The IXA portability framework provides a Control Plane Product Development Kit (CP-PDK), which supports industry standards such as the ones described above and enables data plane software to interface with the Control Plane.

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

SCALING APPLICATIONS TO HIGHER DATA RATES

This section describes how the IXP family of network processors allows an application to be scaled to higher data rates and the challenges involved.

Moving from Single-Chip to Dual-Chip Configurations

A common way to scale an application to higher data rates is to use more microengines running in parallel. The IXP2400 and IXP2800 may be used in both single-chip and two-chip configurations, thereby allowing the developer to increase the number of microengines available.

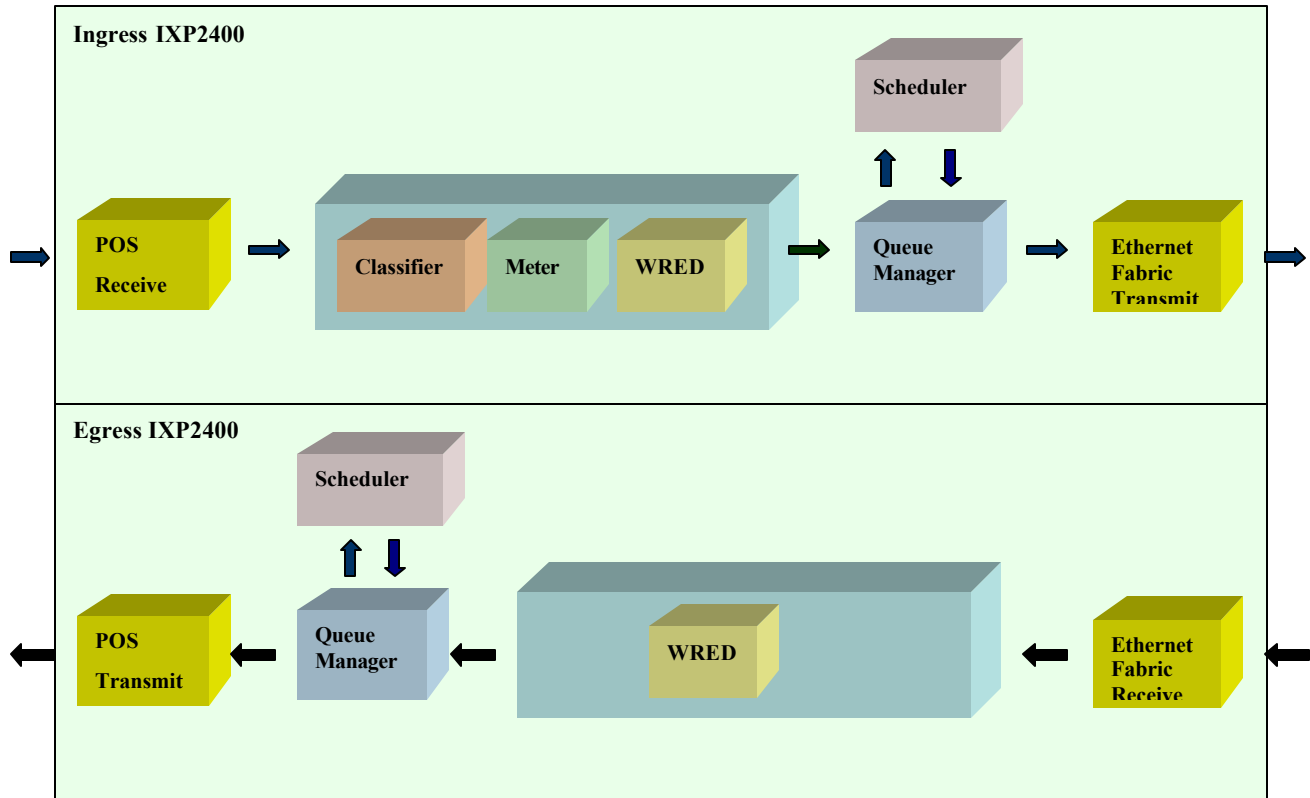


Figure 6: OC-48 DiffServ with dual-chip IXP2400

Figure 6 shows the implementation of DiffServ at OC-48 data rates using two IXP2400 processors. The ingress processor receives from a POS network interface and transmits into an Ethernet fabric. The egress processor receives from the Ethernet fabric and transmits into the POS network interface.

The same application may be run on a single IXP2400 chip at OC-24 data rates, simply by combining the pipelines on the ingress and egress processors onto a single processor. For example, in the single-chip pipeline a single microengine would implement both the POS receive and transmit processing using four threads for each function.

Challenges

There are some limitations to this approach that must be addressed and overcome at design time.

Combining blocks onto a single microengine may not always be possible due to conflicts in usage of resources like CAM and local memory. For example, it may not be possible to combine two schedulers onto a single microengine because both make extensive use of local memory. Similarly, combining the ingress and egress queue managers may not be possible since both use the CAM. However, the developer can combine the scheduler and the queue manager blocks

into a single microengine, since the scheduler does not use the CAM and the queue manager does not use too much local memory.

Another possible problem while combining blocks onto a single microengine is to run out of instruction code store. Again, the only solution is to intelligently pick and combine blocks for which this is not an issue.

Moving from IXP2400 to IXP2800

While the IXP2400 can handle up to OC-48 data rates, the IXP2800 provides the upgrade path to 2 x OC-48 with a single-chip configuration and OC-192 with a dual-chip configuration.

Moving an application from IXP2400 to IXP2800 is greatly simplified by the fact that the instruction set and microengine architecture are identical. The IXP2800 runs at a higher clock frequency (1.4GHz versus 600MHz for the IXP2400) and supports a greater number of microengines (16 versus 8 on the IXP2400).

. Table 1: Comparison of the IXP2400 and IXP2800 for a DiffServ application

Parameter	IXP2400 single chip	IXP2400 dual chip	IXP2800 single chip	IXP2800 dual chip
Line rate	1.2 Gbps	2.408 Gbps	2 x 2.408 Gbps	4x2.408Gbps
Min POS packet size	49	49	49	49
Packet throughput	3.071 million packets/sec	6.14 million packets/sec	12.28 million packets/sec	24.57 million packets/sec
Clock frequency	600 MHz	600 MHz	1.4 GHz	1.4 GHz
Inter-packet arrival time	195.37 cycles	97.68 cycles	114 cycles	57 cycles
Compute cycles per packet for a context pipe stage (microengine)	195.37 cycles	97.68 cycles	114 cycles	57 cycles
Latency per packet for a context pipe stage (microengine)	195.37*8 cycles	97.68*8 cycles	114*8 cycles	57*8 cycles
Compute cycles per packet for a functional pipeline of n microengines running in parallel	195.37*n cycles	97.68*n cycles	114*n cycles	57*n cycles
Latency per packet for a functional pipeline of n microengines running in parallel	195.37*n*8 cycles	97.68*n*8 cycles	114*n*8 cycles	57*n*8 cycles

Table 1 compares the per-packet instruction count and latency constraints for an IXP2400 targeting single-chip 2xOC-12 and dual-chip OC-48 versus an IXP2800 targeting single-chip 2xOC-48 and dual-chip OC-192.

The table indicates that an IXP2800 running in single-chip mode has approximately the same number of instruction cycles (114 cycles) to process a minimum-size POS packet as an IXP2400 in a dual-chip configuration (97 cycles).

CHALLENGES

While the code reuse between applications running at the different data rates is good, there are some limitations that must be considered at design time when moving from an IXP2400 to the IXP2800.

Some blocks cannot be run in parallel on more than one microengine. So the availability of more microengines does not help. The queue manager is an example of such a block. It manages the queuing hardware on the IXP2800 using the CAM local to the microengine. Since it cannot be executed in parallel on more than one microengine, the code for the dual-chip IXP2800

This, along with the fact that a single-chip IXP2800 has the same number of microengines as a two-chip IXP2400 (Table 2), implies that at least at an instruction count level, an application running at OC-48 rates on a dual-chip IXP2400 will scale to 2xOC-48 rates on a single-chip IXP2800. Similar considerations may apply to the OC-192 application running on a dual-chip IXP2800 with 32 available microengines.

must be optimized to fit the OC-192 (57 cycle) instruction budget.

Even though the IXP2800 runs at a higher clock frequency, the memory speed does not scale up the same way. This implies that memory table accesses (relative to the number of instructions executed) take longer on the IXP2800.

There are some differences in the Media Switch Fabric (MSF) between IXP2400 and IXP2800. The differences are minor and may be handled with processor-specific switches in the receive and transmit code.

Table 2: Microengine allocation for the DiffServ application

Function	IXP2400 single chip	IXP2400 dual chip	IXP2800 single chip	IXP2800 dual chip
POS receive	½	1	1	2
POS transmit	½	1	1	2
Classifier/meter/WRED	2	4	6	8
Ingress queue manager	½	1	1	1
Egress queue manager	½	1	1	1
Ingress scheduler	½	1	1	2
Egress scheduler	½	1	1	2
Egress WRED block	½	1	2	4
Ethernet fabric receive	½	1	1	2

The receive and transmit blocks run on a single microengine for the OC-48 and 2 x OC-48 cases, whereas they are executed on two microengines in the OC-192 case. This implies that some of the receive or transmit context stored in local memory may need to be flushed out to SRAM in the OC-192 case.

The implication of these challenges is that while it is very easy to get code written for an IXP2400 to work on an IXP2800, special optimizations may be needed to get the maximum performance from the chip.

CONCLUSION

When network equipment vendors select a network processor, they make a significant commitment to use it for years to come. It is important to ensure that the network processor environment selected is flexible and can be scaled to protect the vendor's investment. Software reusability and the tools and framework that enable it should be a key consideration when selecting a network processor.

The IXP family of network processors provides a powerful and scalable solution to the problem of programmable network devices. The IXA portability framework provides the associated software infrastructure to help develop modular and reusable software building blocks for these processors.

By providing the necessary infrastructure to help accelerate software development and by improving code reuse across applications, the IXA framework adds considerable value to Intel's network processor solution.

ACKNOWLEDGMENTS

The authors acknowledge the contributions of Ameya Varde, Senthil Nathan, Eswar Eduri, and Sridhar Lakshmanamurthy.

REFERENCES

- [1] Matthew Adiletta, et al., "The Next Generation Family of Intel Network Processors," *Intel Technology Journal*, Vol. 6 issue 3, August 2002.
- [2] Matthew Adiletta, et al., "Packet over SONET: Achieving 10 Gigabit/sec Packet Processing with an IXP2800," *Intel Technology Journal*, Vol. 6 issue 3, August 2002.
- [3] S. Blake, et al, "An Architecture for Differentiated Services," IETF RFC 2475, December 1998.

AUTHORS' BIOGRAPHIES

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 Fremont, California, and can be reached via e-mail at uday.naik@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.

Prashant Chandra is a senior staff network architect in the Network Processor Division at Intel Corporation. His interests are in the areas of programmable networks, signaling protocols, and traffic management. He received his B.E degree in Electronics Engineering from Bangalore University in 1991, an M.S. degree in Computer Engineering from West Virginia University in 1994, and a Ph.D. degree in Computer Engineering from Carnegie Mellon University in 2000. His e-mail is prashant.chandra@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. credits finishing at UCLA. He resides in Shrewsbury Massachusetts. His e-mail is donald.hooper@intel.com.

Travis Schluessler is an engineering project lead at Intel's Network Processor Division. Professional interests include networking and embedded systems. Travis received his Bachelor's degree in Electrical and Computer Engineering from Carnegie Mellon University in 1992. He resides in Cupertino, California, and can be reached via e-mail at travis.schluessler@intel.com.

Adrian Georgescu is a staff network software engineer at Intel's Network Processor Division. Adrian received a Master's degree from Polytechnic Institute, Bucharest, Romania, in Aerospace Engineering. His professional interests are networking, OO programming, and numerical algorithms. Currently, he lives in Palo Alto, California, and can be reached by e-mail at adrian.georgescu@intel.com.

Duke Tallam is a software engineering manager in the Network Processor Division at Intel Corporation. Duke has over 20 years of software development experience in a wide variety of fields spanning deeply embedded to huge

turnkey systems. Duke received his Master's degree in Electrical Engineering from South Dakota School of Mines and Technology in 1983. He holds a Bachelor's degree in Electronics from Bangalore University. Duke resides in Fremont, California, and can be reached via e-mail at duke.tallam@intel.com.

Alex Shoykhet is a senior software product manager at Intel's Network Processor Division. He can be reached at alex.shoykhet@intel.com.

Raj Yavatkar is chief software architect for Intel's Network Processor Group. He can be reached at raj.yavatkar@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