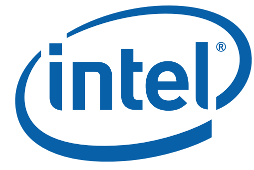
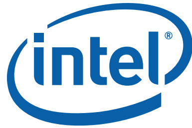


White Paper
Cristian F. Dumitrescu
Software Engineer
Intel Corporation

Design Patterns for Packet Processing Applications on Multi-core Intel® Architecture Processors

December 2008

321058



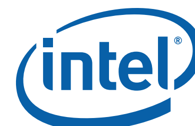
Executive Summary

This paper describes several key design aspects for consideration when implementing packet processing applications on multi-core Intel® architecture processors, presenting the inherent design problems and exploring solutions for each.

This paper aims to help the developers seeking a methodology for how to optimally use the powerful Intel multi-core processors for packet processing when making the transition from NPUs or ASICs to multi-core Intel® architecture processors or when developing new applications

The paper briefly enumerates the benefits of using the multi-core Intel® architecture processors for packet processing and includes a concise comparison between the dual-core and quad-core Intel® architecture processors targeted for the development of networking applications. Main topics include:

- Partitioning the cores between the control and data planes
- Role of the Operating System for the control and data planes
- Programming models for data plane processing: the pipeline model and the cluster model with its two flavors of run-to-completion and request-based.
- Working with accelerators to minimize/hide the latency associated with complex operations specific to packet processing
- Using the Intel® QuickAssist Technology for leveraging the existent industry support for specialized accelerators for packet processing
- Data structure design and inter-core synchronization



Contents

Executive Summary 2

Introduction 4

Why Use Multi-core Intel® Architecture Processors For Packet Processing Applications?4
Intel® Dual-core and Quad-core Processors Overview 5

Control Plane vs. Data Plane: Core Partitioning 6

Control Plane vs. Data Plane: Role of the Operating System 8

Operating System for the Control Plane Cores 8
Operating System for the Data Plane Cores 8

Data Plane Cores Organization and Programming Models: Pipeline and Cluster Models 9

The Pipeline Model 9
The Cluster Model 10

Data Plane: Minimizing/Hiding the Latency of Complex or I/O Intensive Operations 12

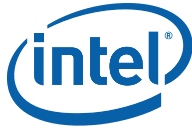
Problem Description 12
Problem Solutions 12

Data Plane: Working With Accelerators 14

Using Intel® QuickAssist Technology 14
Dedicating Specific Cores to Implementing Accelerators 14
Flavors of the Cluster Model 15

Data Plane: Data Structures and the Inter-core Synchronization Problem 17

Conclusion 18



Introduction

With the advent of the latest Intel multi-core processors, it has become feasible from the performance as well as from the power consumption point of view to build complete packet processing applications using the general purpose Intel® architecture processors.

Architects and developers in the industry are now considering these processors as an attractive choice for implementing a wide range of networking applications, as performance levels that could previously be obtained only with network processors (NPUs) or ASICs can now also be achieved with multi-core Intel® architecture processors, but without incurring the disadvantages of the former.

Why Use Multi-core Intel® Architecture Processors For Packet Processing Applications?

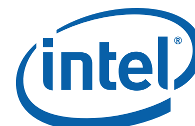
This section discusses some of the reasons that support the choice of making the transition from NPUs or ASICs to multi-core Intel® architecture processors. Some of them are instant benefits brought in by simply using the Intel® architecture processors while others come as consequences of removing the constraints attached to the previous platforms.

Intel® architecture processors are recognized for performance

- Historically, Intel has a proven track record of delivering on the user's expectation for continuous improvement in computing power.
- Intel has disclosed a roadmap for multi-core development that is unmatched throughout the industry and provides a guarantee for long term support.

Intel® architecture processors are recognized for application development support

- Implementing packet processing applications on the Intel multi-core processors enables the reuse of the extensive code base already developed for the Intel® architecture processors including BIOS, all the major Operating Systems, libraries and applications.
- The same statement applies for the mature software development tools already in place for the Intel® architecture processors.
- Intel® architecture is one of the most widely used processor architectures. Developing packet processing applications on Intel® architecture would allow the reuse of the important assets of developer skills and knowledge base on the Intel® architecture processors. As opposed to NPUs, the software engineers are not



required to learn a special purpose programming model and tools for packet processing; instead, they can continue to use the same architecture and tools they are comfortable with and are most productive with.

Intel® architecture processors are recognized for programmability

- As opposed to NPUs which often use special purpose instruction sets, the IA processors have a general purpose architecture and instruction set, which represents the key advantage responsible for their programmability.
- As result of this, the multi-core Intel® architecture processors allow the implementation of highly programmable control and data planes, with fewer constraints than the NPUs. The multi-core Intel® architecture processors offer scalability through software for the control and data plane processing, which is the way to go for networking applications relying on protocols and standards in a continuous evolution.
- Unlike NPUs, the multi-core Intel® architecture processors do not rely on expensive resources that are difficult to scale up, like on-chip multi-port memory, CAM memory, large external SRAM memory, etc.

Intel® Dual-core and Quad-core Processors Overview

Ideally, a single core processor should be powerful enough to handle all the application processing. However, a single core cannot keep up with the constant demand for ever increased computing performance.

The impact of improving the core internal architecture or moving to the latest manufacturing process is limited. Higher clock frequencies also results in considerably higher energy consumption and further increase in the processor-memory frequency gap.

A way to move forward to continue delivering more energy-efficient computing power is to make use of the advantages of parallel processing. In fact, Intel® multi-core chips deliver significantly more performance while consuming less energy. This approach is highly effective for applications such as packet processing.

The latest multi-core Intel® processors are represented by the Intel® Core™ i7 processor family which uses the Intel® Hyper-Threading Technology to combine the advantages of multi-processing with those of multi-threading. Table 1 summarizes the main features of the latest Intel® multi-core processors. To find more about the Intel® multi-core processors, please go to http://www.intel.com/multi-core/.

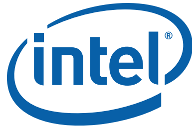


Table 1. Comparison between selected multi-core Intel® processors

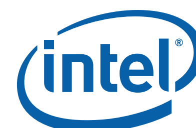
Feature	Intel® Core™ i7 Processor Family	Intel® Core™ 2 Extreme Processor Family	Intel® Core™ 2 Quad Processor Family	Intel® Core™ 2 Duo Processor Family
Selected processor numbers	i7-920, i7-940, i7-965	QX9650, QX9770, QX9775	Q8200, Q8300, Q9300, Q9400, Q9450, Q9550, Q9650	E8200, E8300, E8400, E8500, E8600
Cores	4	4	4	2
Threads per core	2	1	1	1
Processor technology	45 nm	45nm	45 nm	45 nm
Clock speed	2.66 GHz – 3.20GHz	3 GHz – 3.20 GHz	2.33 GHz – 3 GHz	2.66 GHz – 3.33 GHz
Last level cache	8 MB	12 MB	4 MB – 12 MB	6 MB
Intel® QuickPath Interconnect	4.8 – 6.4 GT/s	N/A	N/A	N/A
Front Side Bus speed	N/A	1333 MHz – 1600 MHz	1333 MHz	1333 MHz
Integrated Memory Controller	3 Channels/2 DIMMS/Ch	N/A	N/A	N/A
Memory type/speed	DDR3 800/1066 MHz	DDR2, DDR3	DDR2, DDR3	DDR2, DDR3

Source: compare.intel.com

Control Plane vs. Data Plane: Core Partitioning

The multi-core Intel® architecture processors are designed to serve as vehicles for the development of complete networking applications. The full processing required by the control plane as well as the data plane can be implemented on the same multi-core Intel® architecture chip, although in terms of computing the two network planes have completely opposite sets of requirements.

The data plane, also called the forwarding plane or the fast path, handles the bulk of the incoming traffic that enters the current network node. It is processed according to the rules identified during the classification stage and is sent back to the network. The packet processing pipeline typically includes



stages like parsing, classification, policing, forwarding, editing, queuing and scheduling.

In terms of computing, the data plane is synonymous with real time packet processing. The real time constraints are due to the fact that the amount of processing applied per packet needs to fit into the packet budget, which is a direct consequence of the input packet rate. In other words, each stage along the pipeline must apply its processing on the current packet before the next packet in the input stream arrives; if this timing constraint is not met, then the processor must start dropping packets to reduce the input rate up to the rate it can sustain.

Due to the tight packet budget constraints, the processing applied per packet needs to be straightforward and deterministic upfront. The number of different branches which can be pursued during execution should be minimized, so that the processing is quasi-identical for each input packet. The algorithm should be optimized for the critical path, which should be identified as the path taken by the majority of the incoming packets.

In contrast with the data plane, the control plane is responsible for handling the overhead packets used to relay control information between the network nodes. The control plane packets destined to the current node are extracted from the input stream and consumed locally, as opposed to the bulk of the traffic which is returned back to the network.

The reception of such packets is a rare event when compared with the reception of user packets (this is why the control plane packets are also called exception packets), so their processing does not have to be real time. When compared to the fast path, the processing applied is complex, as a consequence of the inherent complexity built into the control plane protocol stacks, hence the reference to this path as the slow path.

As the processing requirements associated with the two network planes are so different, it is recommended practice that the cores dedicated to data plane processing be different than those handling the control plane processing. As the application layer requires the same type of non-real time processing as the control plane, it usually shares the same cores with the latter.

When the same cores handle both the data plane and the control plane/application layer processing, a negative impact on both may be observed. If higher priority is given to the control plane against the data plane, the handling of the input packets is delayed, which leads to lengthy packet queues as result of network interfaces keeping them well supplied with packets received from the network, which eventually ends up in congestion and packet discards.

If instead the data plane gets higher priority than the control plane, then the delay incurred in handling the hardware events (e.g. link up/down) or the control plane indications (e.g. route add/delete) results in analyzing them when they are already obsolete (the link that was previously reported down might be up by now). This behavior usually has an impact on the overall quality of the system (packets are still getting discarded, although a route for

them is pending for addition) and results in a non-deterministic system with hidden stability flaws.

Control Plane vs. Data Plane: Role of the Operating System

Operating System for the Control Plane Cores

It is standard practice to have the cores allocated to control plane/application layer running under the control of an operating system, as these tasks do not have any real time constraints attached to them with regard to packet processing. In fact, the complex processing which has to be applied and the need to reuse the existing code base make the interaction with the OS a prerequisite.

Historically, the Intel® architecture cores have benefited from excellent support from all the major OS vendors and their programmability and computing performance have recommended them as an excellent choice for the development of complex applications.

Operating System for the Data Plane Cores

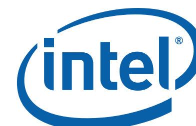
On the other hand, there are strong reasons to discourage the use of an OS for the cores in charge of the data plane processing.

First of all, no user is present, so there is no need to use an OS to provide services to the user or to restrict the access to the hardware. One of the important OS roles is to regulate the user's access to hardware resources (e.g. device registers) through user space / kernel space partitioning. Typically, the operating system allows the user application to access the hardware only though a standardized API (system calls) whose behavior cannot be modified by the user during run-time.

The user does not need to interact directly with the fast path, as the packet forwarding takes place automatically without any need for user's run-time input. The user might influence the run-time behavior of the fast path indirectly through interaction with the control plane by triggering updates of the data structures shared between the fast path and the slow path, but the code that updates these data structures is typically kernel code running on the control plane cores, which does already have full access to hardware.

Secondly, the main functions typically handled by an OS are not required:

- Process management is not required, as there is only one task, which is very well defined: the packet forwarding. Even if a programming model with several tasks synchronizing between themselves would be imagined, the cost of task scheduling in terms of processor cycles would be prohibitively expensive and would severely impact the packet budget with no real value added in return.



- Memory management is usually very simple, as it relies on the usage of pre-allocated buffer pools with all the buffers from the same pool having the same size. There is usually no need to support the dynamic allocation/release of variable size memory blocks, as implemented by the classical malloc/free mechanism.
- File management is not required, as typically there is no file system.
- Device management is usually done through the use of low-level device API functions. The set of existing devices is fixed (network interfaces, accelerators) and small, there is no need to discover the peripherals at run-time or to support hot-pluggable devices. As there is little commonality among the fast path devices, there is little practical gain in implementing a common device interface or a device file system.

Sometimes, out of pure convenience or due to the need to support legacy code, an OS might also be used for the data plane cores. In this case, it might be useful to use a mechanism called *para-partitioning* to create two separate partitions for the control plane and the data plane respectively. This mechanism requires firmware support to partition the resources of a single physical system into multiple logical systems while still maintaining a 1:1 mapping between the logical and the physical resources. Each partition boots its own OS which is aware only of the resources statically assigned to it by the firmware.

Data Plane Cores Organization and Programming Models: Pipeline and Cluster Models

This section explores several programming models for the packet processing pipeline and their influence on performance in the multi-core Intel® architecture processor environment.

The Pipeline Model

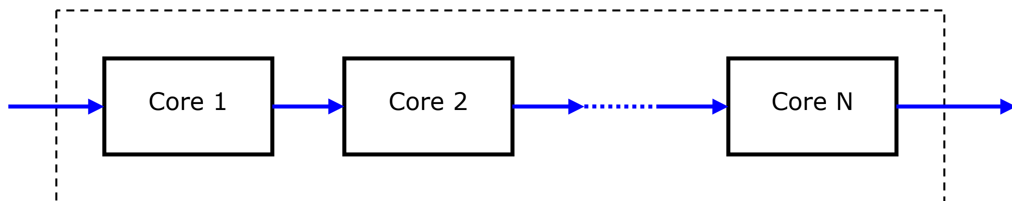
In this model, each stage of the packet processing pipeline is mapped to a different core/thread, with the packet being sent from one stage to the next one in the pipeline. Each core/thread has its fixed place in the pipeline and is the owner of a specific functionality which it applies on a single packet at a time, so the number of packets currently under processing is equal to the number of pipeline stages.

It is often the case that certain functionalities require more processing power than a single core can provide. The way to work around this problem in the pipeline model is to split the associated processing over several pipeline stages, each one executing just a subset of the operations of the initial stage.

The pipeline model has the advantage of offering an easy way to map the packet processing pipeline to the computing resources of the processor by simply assigning each pipeline stage to a different core/thread. However, the disadvantages of this model are significant:

- Waste of computing resources due to their fragmentation. The functionalities are apportioned into pipeline stages so that each stage fits into a core/thread. The fit is not perfect and there is always some computing power headroom left unused which may result in a significant waste overall.
- Expensive memory accesses for packet descriptor caching. After processing the current packet, the current pipeline stage needs to send the packet descriptor to the next stage inline. This requires the current stage to write the descriptor back to the shared memory space at the end of processing and the nest stage to read the descriptor to local memory/registers at the beginning of the processing. If the shared memory is external, then this approach results in expensive memory accesses that may have a severe impact on the packet budget of each of the pipeline stages. If an internal shared memory exists, the access to this memory may constitute a bottle-neck, as all the cores/threads running in parallel are trying to read/write data from the same memory.

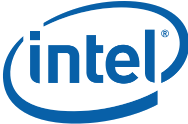
Figure 1. The Pipeline Model



The Cluster Model

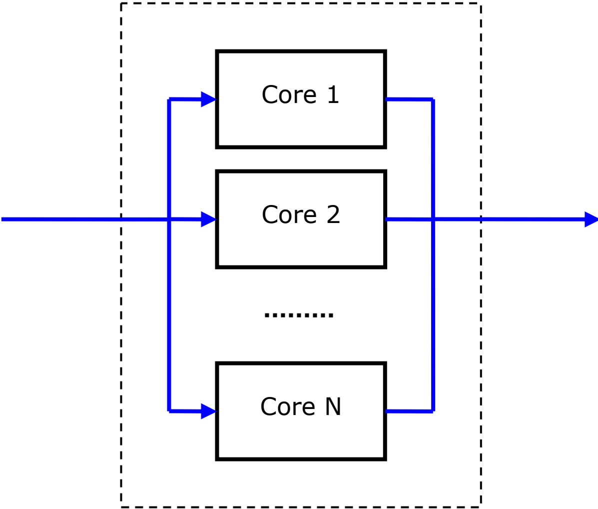
The disadvantages of the pipeline model are addressed by the cluster model, which combines several cores/threads into a cluster running the full (i.e. not fragmented) functionality of the packet processing pipeline. All the cores/threads within the cluster execute the same software image on packets read from the same input stream and written to the same output stream.

From the outside, the number of cluster members is transparent and therefore the cluster looks like a single super-core. The full functionality of the packet processing pipeline is applied on the input packets by a single stage (the cluster) as opposed to sending the packets from one stage to another. However, the cluster model is not problem-free, as it introduces the delicate problem of synchronization between the cluster members when



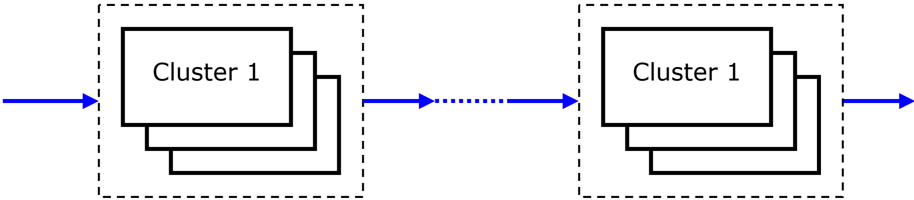
accessing the shared resources of the cluster like the input/output packet streams, the shared data structures, etc.

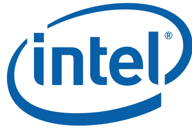
Figure 2. The Cluster Model



Of course, the hybrid model is also possible, in which case the advantages of both models are combined by mapping the packet processing pipeline to a pipeline of interconnected clusters.

Figure 3. The Hybrid Model: Pipeline of Interconnected Clusters





Data Plane: Minimizing/Hiding the Latency of Complex or I/O Intensive Operations

Problem Description

Apart from common operations performed by any application, the packet processing also involves some specific operations that cannot be efficiently implemented with a general purpose instruction set. Typical examples include:

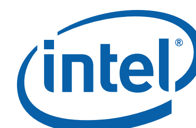
- Table lookup using algorithms like Longest Prefix Match (LPM) or range matching, required during the classification or forwarding stages of IPv4 routing applications.
- Pattern matching, used by deep packet inspection applications that filter the stream of incoming packets by applying regular expressions stored in the rule data-base over the packet payload.
- Cryptography, used by security applications to decrypt/encrypt the payload of incoming/outgoing packets.
- CRC or checksum calculation, used by many networking protocols to indicate the integrity of the packets received from the network.

The size of the available on-chip memory is usually small, if any. As a result, some operations require a significant number of accesses to external memory. Although several techniques are available to minimize their number (e.g. by using on-chip cache memories) or optimize them (e.g. by using DMA engines or memory bank interleaving), the penalty incurred by these accesses cannot be completely eliminated. Their impact on the packet budget is significant, and sometimes the memory accesses per packet can even exceed the budget of processor cycles per packet. Examples of such operations are:

- Reading the packet descriptor to local memory and writing it back to external memory (header caching).
- Software implementations of table lookup algorithms.

Problem Solutions

To work around these two problems, several measures can be considered. Some of them are described below.



Use the advantages of parallel processing to reduce the pressure of meeting packet budget

When using the pipeline model, each stage of the pipeline must still meet the packet budget, but the amount of processing that needs to fit into this number of clock cycles is limited to the one of the current pipeline stage as opposed to the full packet processing implemented by the entire system. As a result of each pipeline stage processing a different packet, the packet budget is effectively multiplied with the number of pipeline stages.

When using the cluster model, the cluster is still required to process one packet per each packet budget interval, but by using internally several cores running in parallel, each one processing a different packet, the packet budget is effectively multiplied with the number of cores within the cluster.

Use multi-threading to hide the latency of accesses to external memory

No increase in the processing power is achieved by simply using the multi-threading. As the threads run in time sharing mode, all threads running on the same core use up a share of the processing cycles of their core. However, multi-threading can be an important instrument in minimizing the waste of processing cycles of the core.

In a cooperative multi-threading system, the current thread can decide to relinquish the control of the core to other threads immediately after sending a request to the external memory controller. The memory access takes place while the thread which launched it is dormant and other threads are using the core to perform meaningful work on other packets. The thread will not resume the processing of the packet until the memory access has completed, thus the core is not wasting processing cycles as it does not busy-wait for the memory access to complete.

Therefore, as no increase in the processing power is achieved, multi-threading cannot minimize the latency of complex operations, but it can be an effective mechanism to *hide* this latency from the cores and thus increase the overall efficiency of the cores.

Use specialized accelerators to offload the complex or memory intensive operations from the processing cores

One of the roles of accelerators is to reduce the latency of the complex or memory intensive operations. For example, by using a specialized hardware block, the decryption (or encryption) of the current packet can be performed in just a few clock cycles, which is not possible when implementing the same operation with a general purpose instruction set. The complex mathematical operations associated with cryptography, if not offloaded to a specialized engine, can effectively choke the packet processing cores.

The other role that may be assigned to accelerators is to hide the latency of the I/O or memory intensive operations from the cores in charge of running

the packet processing pipeline. The latency of the operation is not reduced, but by offloading it to a specialized block, the cores are given the chance to do some other useful work, i.e. process some other packets meanwhile and resume the processing of this packet after the result from the accelerator becomes available.

This approach aims at maximizing the core utilization and thus minimizing the number of necessary cores. The alternative would be to block while busy-waiting for the memory transfers to complete, which would require adding more cores/threads just to keep up with the incoming packet stream, which results in an inefficient setup due to big idle time percentage for the cores.

Data Plane: Working With Accelerators

Using Intel® QuickAssist Technology

To facilitate the integration of the various solutions for specialized accelerators currently offered by the industry with the Intel® architecture processors, Intel has introduced the Intel® QuickAssist Technology. This represents a design framework that standardizes both hardware and software interfaces to enable portability from one hardware platform to another while requiring minimal modifications to the application software.

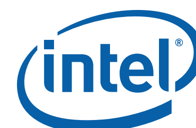
On the hardware side, this framework specifies the interfaces for connecting the accelerators as IP blocks on the same die with Intel® architecture cores, as well as connecting external accelerators with the Intel® architecture processors through the PCI Express*, Front Side Bus or Intel® QuickPath Interconnect interfaces.

From the software perspective, the well defined APIs have the role of abstracting the underlying hardware implementation of accelerators, including the connectivity method to the Intel® architecture processors, so that upgrading to a new platform does not impact the software application.

The first Intel® architecture chip that is Intel® QuickAssist Technology enabled is the Intel® EP80579 Integrated Processor, who provides accelerated on-chip support for encryption/decryption, as well as other packet and voice processing tasks. To find out more about the Intel® QuickAssist Technology, please go to: http://www.intel.com/technology/platforms/quickassist/.

Dedicating Specific Cores to Implementing Accelerators

When accelerators are not available to offload a complex operation from the processing cores, it is always a good practice to segregate the code that implements this operation from the code that deals with regular processing



and assign them to different cores. The cores are basically divided in two separate categories: packet processing cores and I/O cores.

The purpose of this strategy is to eliminate the busy-waiting operations from the processing cores by moving them to dedicated cores. The performance of the overall system becomes more predictable, as the number of cores dedicated to I/O processing is a direct consequence of the number of I/O operations per packet, which can be determined upfront. This way, the regular processing is not prevented by busy-waiting from utilizing the core idle time.

The I/O cores can be looked at by the packet processing cores as accelerators, as the I/O cores are effectively implementing complex or memory intensive tasks which are this way offloaded from the packet processing cores. The communication between the two can be implemented with message passing with the packet processing cores (the clients) sending requests to the I/O cores (the servers) and eventually receiving responses from them when the result of the operation becomes available.

One notable example feasible to implement using this approach is the preprocessing of traffic received by the network interfaces, as well as its preparation for transmission. On the reception side, the network interface (e.g. a Gigabit Ethernet MAC) receives a packet from the network, stores it into a memory buffer and writes the buffer handle into a ring read by the processor cores. Building the packet descriptor in a proper format that is understood by the rest of the pipeline might be the job of one or more I/O cores that block while waiting for a new packet to be received by the MAC and implement the low-level handshaking mechanism with the MAC. Similar processing is required on the transmission side.

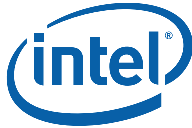
To support higher bandwidth (e.g. 10 Gigabit Ethernet), the Intel® MACs support a feature called Receive Side Scaling (RSS), which applies a MAC built-in hashing function on selected fields of the input packet to uniformly generate the index of an I/O core out of a pool of cores assigned for this purpose. This way, a fair load-balancing between the cores is performed in order to achieve the high input bandwidth of the network interface.

Another good example is the implementation of the table lookup operation used during the classification and the forwarding stages. The software algorithms for LPM require several memory accesses into tree-like data structures for each table lookup. When the I/O core implementing it receives a request to find the LPM match for a specific key (IP address plus prefix), it performs the sequence of memory reads while blocking after each read until completed. When the final result is available, it is returned as the response to the processing cores.

Flavors of the Cluster Model

Run-to-completion Model

This model assumes the existence of multi-threading on the processing cores. Each packet entering the cluster is assigned a single thread that processes



the packet completely, i.e. it handles the packet from the moment it enters the cluster until it exits the cluster. No other thread can process this packet.

In a cooperative multi-threading system, the core applies the regular processing on the packet and sends it to specialized accelerators for complex operations while releasing the control of the processor to other threads processing different packets. The processing of the current packet is resumed by the same thread after the result from the accelerator becomes available.

As the packet descriptor is accessed by a single thread, this model has the advantage that the packet descriptor can be stored in the private memory of the assigned thread. The disadvantage is that the number of packets that can be processed at a given time cannot exceed the number of available threads, so when all the threads of a core are dormant waiting for response from the accelerators, the core cannot read new packets from the input stream even though it is currently idle.

Request-based Model

This model does not require multi-threading. It uses the processing cores as dispatchers between accelerators. Each core reads a different packet, applies the regular processing and sends the packet to specialized accelerators for complex operations. While the accelerator is working, the core goes to process another packet.

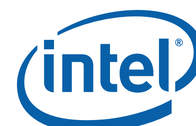
The difference from the previous model is that any core can resume the processing of the current packet once the job of the accelerator is completed. This approach assumes that all the processing cores are sharing the interrupt requests coming from the accelerators.

This model has the advantage that the number of packets under processing is not limited by the number of available threads, but it has the drawback that the packet descriptors must be stored in the shared memory space to make them available to all the cores.

Table 2 summarizes the differences between the two models.

Table 2. Run-to-completion vs. Request-based

Item	Run-to-completion	Request-based
Can the current packet be processed by more than one thread?	No	Yes
Can the number of packets under processing be bigger than the number of threads?	No	Yes
Memory storing the packet descriptor	Thread private memory	Shared memory



Data Plane: Data Structures and the Inter-core Synchronization Problem

The cluster model introduces the problem of synchronization between the cores/threads for accessing the cluster shared resources like the input and output packet streams, the specialized accelerators and the shared data structures. Similarly, the pipeline model also must handle the synchronization problem for accessing the data structures that are shared between multiple pipeline stages.

Several types of data structures can be identified for packet processing:

- Data structures per packet: the packet descriptor.
- Data structures per flow/connection: one entry per flow in various tables like flow tables, policy tables, routing tables, etc.
- Data structures per protocol: global structures storing the current context (i.e. the state information) for the processing state machines associated with every networking protocol implemented by the application, data structures with statistics counters, etc.

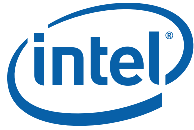
The access to these shared data structures must be protected through semaphores or similar synchronization mechanisms. In general, it is recommended to design the data structures of the application for minimal contention between the cores/threads accessing them.

In some cases, the packet descriptors are not shared between several cores/threads:

- For the pipeline model, if header caching is used
- For the cluster run-to-completion model

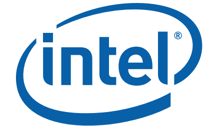
The typical communication mechanism between cores/threads or between cores/threads and the accelerators is message passing using interrupt requests or queues of request/response messages. If no hardware support for queues is available, they can be implemented as linked lists protected by semaphores.

When queues of packets are used as the communication mechanism between the various processing entities, an additional requirement may be to preserve the packet order within the same flow. This means that packets belonging to the same flow/connection should exit the processor in the same order they entered the processor. The most widely used method to achieve this is to assign a unique sequence number to each packet when it enters the processor, allow the packets to be processed out of order for efficiency and have them re-ordered based on their sequence numbers just before their return to the network.



Conclusion

This paper presented several important topics to be considered during the design phase in order to optimize the performance of the packet processing applications running on the Intel® multi-core processors.

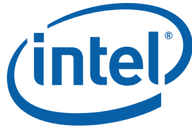


Authors

Cristian F. Dumitrescu is a Software Engineer with the Embedded and Communications Group at Intel Corporation.

Acronyms

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CAM	Content Addressable Memory
CRC	Cyclic Redundancy Check
FPGA	Field Programmable Gate Array
FSB	Front Side Bus
IP	Intellectual Property
IPv4	Internet Protocol version 4
LPM	Longest Prefix Match
MAC	Media Access Control
NPU	Network Processor
OS	Operating System
PCI	Peripheral Component Interconnect
RAM	Random Access Memory
RSS	Receive Side Scaling
SRAM	Static RAM



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

This paper is for informational purposes only. THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel, the Intel logo, Intel. leap ahead. and Intel. Leap ahead. logo, Intel Core i7, Intel Dual-core processors, Intel Quad-core processors, Intel QuickAssist Technology, Intel EP80579 Integrated Processor, Intel Core 2 Extreme Processor, Intel Core 2 Quad Processor, and Intel Core 2 Duo Processor are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2008 Intel Corporation. All rights reserved.