U.P.B. Sci. Bull., Series C, Vol. 73, Iss. 2, 2011 ISSN 1454-234x

# DESIGN PRINCIPLES FOR BUILDING NETWORKING APPLICATIONS USING GENERAL PURPOSE MULTICORE PROCESSORS AND PACKET PROCESSING ACCELERATORS

Cristian F. DUMITRESCU[1]

*Utilizarea procesoarelor multicore cu arhitectură de uz general pentru implementarea aplicațiilor de procesare a traficului din cadrul unei rețele de comunicații nu este o temă uşor de rezolvat. Lucrarea de față examinează unele din problemele cheie care intervin în proiectarea unui astfel de sistem şi explorează spațiul de soluții aflat la dispoziție pentru a stabili un set de principii ce trebuie respectate în activitatea de proiectare. Deoarece una dintre cerințele arhitecturale cheie pentru orice procesor utilizat la procesarea de pachete este o foarte bună programabilitate, lucrarea propune mai multe modele de programare care să facă posibilă procesarea de pachete într-un mediu multicore. Rezultatele analizei sunt folosite pentru a ilustra modul în care aplicația de rutare a pachetelor IPv4, curent folosită în industrie pentru măsurarea performanței de calcul, poate fi implementată pe un procesor multicore de uz general echipat cu acceleratoare pentru procesarea de pachete.*

*Using general purpose multicore processors to build networking applications is not an easy task. This paper examines the key design issues and explores the solution space to identify a set of design principles to address this challenge. As high programmability is one of the key requirements for any packet processing architecture, several programming models are proposed to enable the packet processing workload in the multicore environment. The findings are used to illustrate how the industry standard benchmarking application of IPv4 forwarding can be efficiently implemented on a general purpose multicore processor equipped with packet processing accelerators.*

**Keywords:** networking, packet processing, multicore processors, accelerators, programming model

## 1. Introduction

With the advent of the latest generation of 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 general purpose architecture processors rather than network processors (NPUs) or dedicated Application Specific Integrated Circuits (ASICs).

[1] PhD Student, Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest, Romania, e-mail: cristian.dumitrescu@intel.com

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 NPUs or ASICs are now achievable with multicore processors, but without incurring the disadvantages of the former.

There are several papers that are looking at various hardware and software mechanisms to enabling packet processing on NPU architectures [1], [2], [3]. This paper examines the usage of general purpose multicore processors to build packet processing applications. As high programmability is one of the main reasons for using general purpose architectures for applications previously reserved for NPUs and ASICs, several programming models are proposed here to facilitate the packet processing workload in this environment. One of them, the request-based model, is then used to illustrate how the industry standard benchmarking application of IPv4 forwarding can be efficiently implemented on a general purpose multicore processor equipped with packet processing accelerators.

## 2. Why multicore?

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 result in considerably higher energy consumption and further increase in the processor-memory frequency gap.

The way to go forward to continue delivering more energy efficient computing power is to make use of the advantages of parallel processing. In fact, the latest multicore processors deliver significantly more performance while consuming less energy. This approach is definitely the way to go forward for applications like packet processing.

## 3. Why general purpose architecture processors?

Good programmability is one of the critical requirements for any processor architecture to be successful. Historically, the poor programmability of the NPUs is one of the main reasons these architectures were not successful long term.

To provide functional scalability, the data plane implementation needs to be scalable through software, as the support for additional networking protocols can be implemented much easier/faster in software than in hardware.

The processor should provide good performance for packet processing, but in the same time it also needs to offer the same degree of programmability as the general purpose architectures. Therefore, the processor cores need to have a general purpose rather than specialized architecture and instruction set. Examples of general purpose multicore processor architectures currently available are: Intel® 64, AMD64®, Power Architecture®, MIPS64® or ARMv7®.

The topics of core partitioning between the control plane and the data plane and role of the operating system in a general purpose multicore processor used for packet processing are explored at length in [4].

## 4. Why using packet processing accelerators?

Apart from the operations commonly performed by any application, the packet processing workload also involves some specific operations that cannot be efficiently implemented with the general purpose architecture processor cores. Typically, such operations are either compute intensive (e.g. encryption) or I/O intensive (e.g. external memory intensive operations). In order to meet the packet budget, these operations have to be offloaded from the processor cores to specialized hardware blocks called accelerators.

Each accelerator has one of the following roles:

- *Reduce* the latency of the offloaded operation. Typical example: encryption. By using a specialized accelerator block for this operation it becomes possible to encrypt/decrypt a packet in significantly less clock cycles than a general purpose core;
- *Hide* the latency of the offloaded operation from the cores. Typical example: external memory intensive operations. The latency of the operation cannot be significantly reduced by using an accelerator block instead of a core, but the operation is still offloaded to the accelerator in order to enable the cores to do something useful instead of blocking, i.e. process other packets meanwhile, and come back to the same packet once the accelerator work is completed and the result of the operation is available.

## 5. Building packet processing accelerators

Depending on the specifics of each task, the associated accelerator can be implemented either as a hardwired block or as a programmable block that might optionally drive some task specific hardwired logic.

The main advantage of latter approach is the fact that the same accelerator design can be reused to implement different tasks. The instruction set can be customized for each accelerator, e.g. by removing those instructions that are not necessary for the current task.

The programmable accelerator design should be optimized for multi-threading. Having several hardware threads with hardware supported contexts (allowing zero thread switching overhead) is usually a useful feature to have for such an accelerator. This becomes particularly important for those accelerators implementing memory-intensive operations.

Unlike the processor cores, the accelerator instruction set should contain special purpose instructions supporting the specific task, like: instructions optimized for bit field manipulation, linked list manipulation, etc.

Unlike the processor cores, the accelerators are generally programmed directly in their assembly language, as the compiler overhead is usually not affordable. This is acceptable from the software development productivity point of view, as each accelerator should only be programmed once (firmware) for each task. The fact that an accelerator is programmed instead of hardcoded should be transparent to the software running on the processor cores.

The programmable accelerator design removes the need to build another custom accelerator from scratch for every acceleration task. Most of the hardwired accelerators can be replaced by an array of on-chip programmable accelerators, with the function of each accelerator decided at initialization time by loading the appropriate code image into its instruction memory. Moreover, the processor can be equipped with more programmable accelerators than initially needed in order to accommodate future improvements in the application.

## 6. Data plane programming models.

### The pipeline model

In this model, each stage of the data plane pipeline is mapped to a different processor core or accelerator, with the packet being sent from one stage to the next one in the pipeline. Each block has its fixed place in the pipeline and owns a specific stage that it applies on a single packet at a time.

As each stage is processing a different packet, the packet budget per stage is the overall packet budget (determined by the rate of input packets) multiplied with the number of pipeline stages.

It is often the case that some stages require more processing cycles than can be achieved with just a single instance of the specific functional block. The way to work around this problem is to either break this stage into several stages or use the cluster model internally for this stage.

The pipeline model offers a simple method to map the data plane pipeline to the available set of processor cores and accelerators. The lowlights of the model are:

- Potential waste of computing resources due to their fragmentation when the computing headroom left unused for some of the blocks is significant;
- Potential impact to the memory bandwidth due to the packet descriptors having to be copied from one stage to the next one throughout the pipeline.

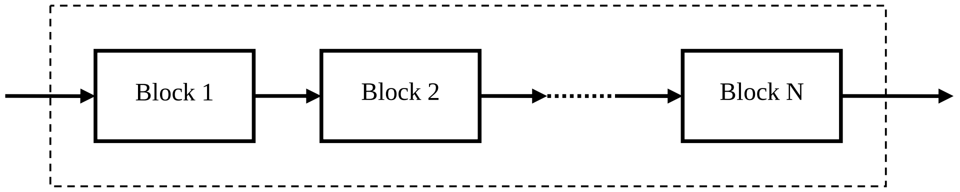

Fig. 1. The pipeline model

**The cluster model**

This model combines several instances of the same functional block, either processor core or accelerator, into a cluster with each cluster member applying the same processing on a different packet. As opposed to the pipeline model, each input packet is completely handled by a single functional block, which is why this model is also called the run-to-completion model.

As each cluster member is processing a different packet, the packet budget per member is the packet budget of the cluster (determined by the rate of input packets) multiplied with the number of cluster members.

From the outside, the number of cluster members is transparent and therefore the cluster looks like a single super-block. However, the cluster model is not free of potential problems:

- The input and output streams of packets are shared by all the cluster members, therefore synchronization between the members is required to serialize their access to the packet streams;
- A mechanism for preserving the packet order within the same connection has to be put in place.

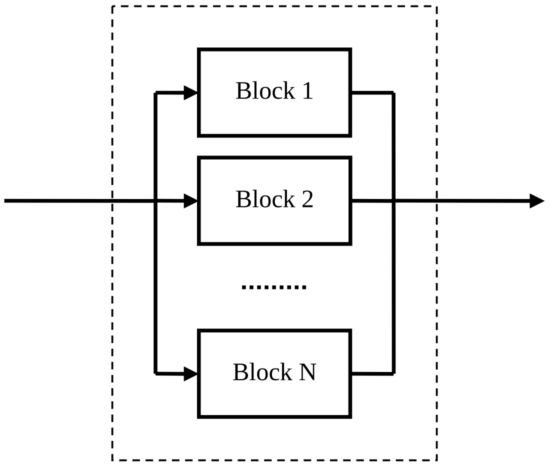

Fig. 2. The cluster model

**The hybrid model**

The hybrid model combines the advantages of both models by mapping the data plane processing to a pipeline of interconnected clusters. One flavor of the hybrid model is the request-based model presented next.

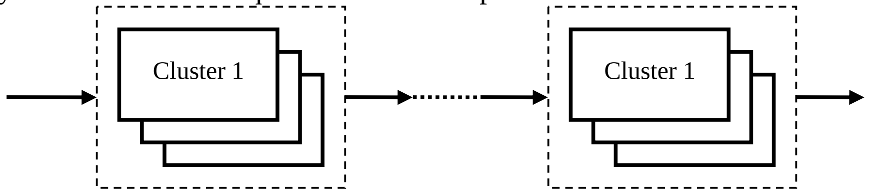

Fig. 3. The hybrid model

**The request-based model**

This model assumes that the processor cores are connected with the accelerators through queues of request/response messages and with the network interfaces through send/receive queues of packet descriptors. In this model, the processor cores are acting as routers of packets from one network interface or accelerator to the next network interface or accelerator in line.

Each processor core is assigned one or more input queues which can be either packet reception queues from the network interfaces or response message queues from the accelerators. All the packets from the same input queue suffer the same processing, which is a characteristic of each input queue. After applying the processing associated with the input queue the packet was read from, the processor core sends the packet to the next accelerator (for further processing) or network interface (for transmission) in line, and then it goes back to scanning its input queues for the next packet to process.

This model organizes the processor cores as a cluster serving a number of input queues and writing packets to several output queues. The packet descriptors have to be stored in a shared memory space to make them available to all the processor cores and accelerators, unless the packet descriptor is small enough to fit the request – response messages that are passed between the internal blocks. If an input queue is scanned by more than one core, a mechanism has to be put in place to enforce the preservation of the packet order for the packets that are part of the same traffic flow/connection.

## 7. Case study: Implementation of the IPv4 forwarding benchmark application

This section illustrates how the design principles presented in this paper can be used to implement the industry de-facto standard benchmarking application of IPv4 forwarding on a multicore processor with on-chip accelerators for packet processing.

This application performs the routing of IPv4 packets according to the IP Classless Inter-domain Routing (CIDR) mechanism [5] in a system with multiple Ethernet interfaces, either Gigabit Ethernet (GbE) or 10 Gigabit Ethernet (10GbE). For each input packet, the output interface is determined by searching through the IPv4 routing table to find the best matching route for the current packet. The search algorithm is Longest Prefix Match (LPM) and the lookup key is the destination IP address read from the IPv4 header of the packet. On lookup hit, the lookup result is the index of the output interface the packet should be sent to.

The packets with a valid route are further subjected to a lookup into the Address Resolution Protocol (ARP) table [6]. The search algorithm is exact match and the lookup key is the same destination IP address. On lookup hit, the lookup result is the destination MAC address that should be stored in the output packet.

The internal architecture of the application is presented in Fig. 4.

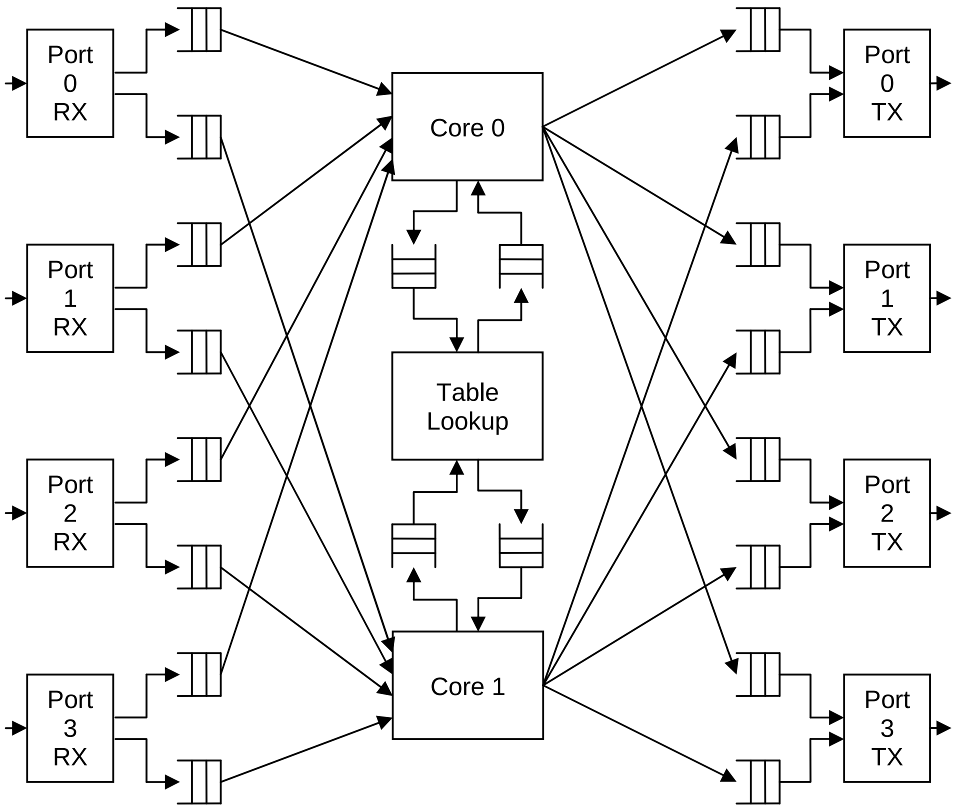

Fig. 4. IPv4 forwarding implemented with the request-based programming model

In this example, there are four 10GbE network interfaces, two processor cores and one accelerator for table lookup, but in general the number of network interfaces, processor cores and accelerator instances that are needed to sustain the rate of input traffic required by the application is determined during the design process.

Each network interface performs the load balancing of the input traffic for the processor cores. The communication between the network interfaces and the processor cores is done through queues of packet descriptors that are written by the network interfaces and read by the cores. Each network interface is connected with each processor core through a separate queue. Similarly, each processor core has its own pair of request-response queues with the table lookup accelerator.

The load balancing logic implemented by each network interface needs to meet several constraints. The first one is to make sure that the input traffic is evenly distributed between its output queues. The second one is to enforce the packet order within each traffic flow. For this application, a traffic flow is uniquely identified by the DiffServ 5-tuple of the following fields read from the input packet: source IP address, destination IP address, transport layer protocol, transport layer source port, transport layer destination port. Examples of commonly used transport layer protocols are User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).

All the input packets with the same 5-tuple are considered to be part of the same traffic flow and therefore the order they exit the processor should match the order they entered the processor. Usually, for Quality of Service (QoS) reasons, all the packets that are part of the same traffic flow follow the same path through the network from source to destination, as set up by the control plane when the traffic flow is initiated. Therefore, all the packets that are part of the same traffic flow enter the processor through the same input interface and are sent out through the same output interface, which simplifies the packet ordering problem by narrowing down its scope to a single receive side interface and a single transmission side interface. Similarly, packet reordering is allowed for packets that are not part of the same traffic flow.

One way to implement the load balancing logic to meet both conditions specified above is to derive the index of the output queue for the current input packet by applying a hash function on the 5-tuple read from the packet and then applying the modulo operator with the number of output queues per interface. The uniform distribution of the hash function ensures that the traffic is spread evenly to the output queues [7], [8], [9]. The use of the 5-tuple as the hash key makes sure that all the packets that are part of the same traffic flow are reaching the same queue. As each queue is read by a single core, the packets from the same traffic flow are processed in the order of their arrival.

The data plane programming model used by this application is request based, which is a flavor of the hybrid model. The packets traverse the following pipeline: network interface reception → processor core cluster → table lookup accelerator cluster → processor core cluster → network interface transmission.

*Table 1*

**Packet budget and processor internal configuration for minimum packet size traffic distribution (all packets are 64 bytes) and processor frequency of 1 GHz**

| | 4x GbE | 8x GbE | 16x GbE | 4x 10GbE | 8x 10GbE | 16x 10GbE |
|---|---|---|---|---|---|---|
| Input rate (Gbps) | 4 | 8 | 16 | 40 | 80 | 160 |
| Input rate (Mpps) | 5.95 | 11.90 | 23.81 | 59.52 | 119.05 | 238.10 |
| Packet budget (ns) | 168 | 84 | 42 | 16.80 | 8.40 | 4.20 |
| Packet budget (cycles at 1GHz) | 168 | 84 | 42 | 16.80 | 8.40 | 4.20 |
| Processor core cycles per packet | 100 | 100 | 100 | 100 | 100 | 100 |
| Number of processor cores | 1 | 2 | 3 | 6 | 12 | 24 |
| Accelerator cycles per packet | 160 | 160 | 160 | 160 | 160 | 160 |
| Number of accelerator instances | 1 | 2 | 4 | 10 | 20 | 39 |

*Table 2*

**Packet budget and processor internal configuration for Cisco IMIX traffic distribution (average packet size of 354 bytes) and processor frequency of 1 GHz**

| | 4x GbE | 8x GbE | 16x GbE | 4x 10GbE | 8x 10GbE | 16x 10GbE |
|---|---|---|---|---|---|---|
| Input rate (Gbps) | 4 | 8 | 16 | 40 | 80 | 160 |
| Input rate (Mpps) | 1.34 | 2.67 | 5.35 | 13.37 | 26.74 | 53.48 |
| Packet budget (ns) | 748 | 374 | 187 | 74.80 | 37.40 | 18.70 |
| Packet budget (cycles at 1GHz) | 748 | 374 | 187 | 74.80 | 37.40 | 18.70 |
| Processor core cycles per packet | 100 | 100 | 100 | 100 | 100 | 100 |
| Number of processor cores | 1 | 1 | 1 | 2 | 3 | 6 |
| Accelerator cycles per packet | 160 | 160 | 160 | 160 | 160 | 160 |
| Number of accelerator instances | 1 | 1 | 1 | 3 | 5 | 9 |

The number of members in the processor core cluster and table lookup accelerator cluster is determined based on the packet budget and the amount of processing performed per packet, as illustrated in *Table 1* and *Table 2*. For example, let us discuss the case of the IPv4 forwarding application with four 10GbE interfaces and Cisco IMIX traffic distribution. This distribution includes 7 small packets (64 bytes), 4 medium sized packets (570 bytes) and a single large packet (1518 bytes), which results in an average packet size of 354 bytes. For four 10GbE interfaces, this average packet size results in an input rate of 13.37 million packets per second (Mpps) and equivalently a packet budget of 74.8 ns or 74.8 cycles for a processor frequency of 1 GHz. Considering that a single processor core needs about 100 cycles to process a packet and a single instance of the table lookup accelerator can handle a packet in about 160 cycles, this packet budget can be met with 2 processor cores and 3 instances of the table lookup accelerator.

## 8. Conclusions

High programmability is one of the most important requirements driving the design of the multicore processors for packet processing. Using general purpose architectures for the processing cores to meet this requirement is feasible as long as the processor is equipped with a set of specialized accelerators to address those operations that cannot be efficiently implemented by the cores.

To fully utilize the power of the cores and accelerators for packet processing workloads, the programming model requires special attention. Several programming models are proposed in this paper: the pipeline model, the cluster model, the hybrid model and the request-based model. The latter is a flavor of the hybrid model, as it combines both the pipeline and the cluster models to build the application.

The request-based model is used to illustrate the implementation of the de-facto industry standard benchmarking application of IPv4 forwarding on a general purpose multicore processor with packet processing accelerators.

## R E F E R E N C E S

[1] *M.A. Franklin, P. Crowley, H. Hadimioglu, P.Z. Onufryc,* "Network Processor Design. Issues and Practices", Morgan Kaufmann Publishers, San Francisco, 2005
[2] *T. Wolf, N. Weng, C.-H. Tai,* "Runtime Support for Multicore Packet Processing Systems", in IEEE Network, July/August 2007, pp. 29-37
[3] *T.L. Riche, J. Mudigonda, H.M. Vin,* "Experimental evaluation of load balancers in packet processing systems", in Workshop on Building Block Engine Architectures for Computers and Networks, 2004
[4] *C.F. Dumitrescu*, “Design Patterns for Packet Processing Applications on Multi-core Intel Architecture Processors”, Intel Design Center, http://edc.intel.com/Link.aspx?id=1187
[5] RFC 4623 “Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan”, www.ietf.org
[6] RFC 826 “An Ethernet Address Resolution Protocol”, www.ietf.org
[7] Bob Jenkins hash function for table lookup (jhash), http://burtleburtle.net/bob/c/lookup3.c
[8] Paul Hsieh hash function (SuperFastHash), http://www.azillionmonkeys.com/qed/hash.html
[9] Fowler, Noll, Vo (FNV) hash function, http://www.isthe.com/chongo/tech/comp/fnv