Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Computer Networks 41 (2003) 587–600

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Fundamental architectural considerations for network processors

Mohammad Peyravian *, Jean Calvignac

IBM Corporation, P.O. Box 12195, Research Triangle Park, NC 27709, USA

Abstract

Network processors (NPs) are programmable devices with special architectural features that are optimized to perform packet-processing functions. They have emerged to cope with the ever-changing networking applications that are becoming increasingly complex. NPs are expected to become the silicon core of network equipments that require a high degree of flexibility to support evolving network services at extraordinary performance with high packet rates. In this paper, we present and examine various NP architectural aspects. We describe and compare NP design characteristics and analyze their implications on the ease of programming.

Keywords: Architecture; Classification; Multiprocessing; Multithreading; Network processor; Packet processing; Programming; Traffic management

1. Introduction

The quest for intelligent and flexible packet processing at high speeds has led to the creation of network processors (NPs). NPs are becoming the silicon core of network equipments that require a high degree of flexibility to support evolving network services at extraordinary performance with high packet rates [4]. Whereas in the past networking equipments were based either on general-purpose processors (GPPs) or application specific integrated circuits (ASICs), favoring flexibility over speed or vice versa, the NP approach achieves both flexibility and performance. The key advantage of NPs is that hardware-level performance is complemented by flexible software.

NPs are programmable devices with special architectural features that are optimized for packet processing. They are designed to perform the common networking functions above the physical layer. From a functionality point of view, network processing can be divided into two general categories: control-plane and data-plane (Fig. 1). Each category has different characteristics and performance requirements.

Typical control-plane protocols are not performance-critical and have modest performance requirements. Examples of control-plane protocols include the resource reservation protocol (RSVP) which is used to allocate resources in routers for IP flows and the open shortest path first (OSPF) protocol which is used to establish and update routing tables. Control plane protocols are best

* Corresponding author. Tel.: +1-919-254-7576.
E-mail address: peyravn@us.ibm.com (M. Peyravian).

1389-1286/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S1389-1286(02)00451-6

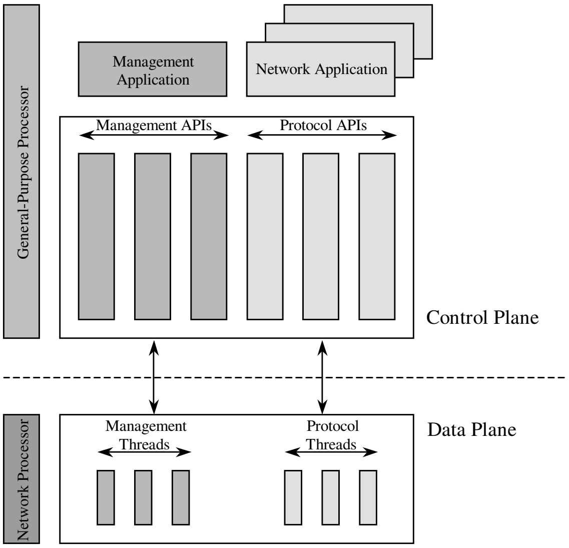


Fig. 1. Network processing breakdown.

suited for GPPs since they have long code paths and exhibit little data parallelism. Control-plane processors (such as PowerPC) are used to process control packets which help in performing data-plane functions. The control plane processor is also in charge of the overall system management.

Data-plane protocols are responsible for forwarding packets. IP-packet and ATM-cell forwarding are examples of data-plane protocol processing. Data-plane processing is performance-critical since it must be performed at high speed to avoid dropping packets and to meet quality of service (QoS) requirements. Data-plane protocols are best suited for parallel processors since they have short code paths and exhibit large data parallelism. Most NPs are optimized to implement data-plane packet processing functions.

The distinction between data- and control-plane protocols becomes blurry as one moves up the protocol stack. For example, the transport control protocol (TCP), a layer 4 protocol which provides a reliable communication path for higher layer protocols, exhibits both data- and control-plane functions and has a rather long code path. This along with the evolving deep-packet processing requirements make designing flexible and cost-effective NPs suitable for layers 2–7 processing an increasingly challenging task.

In the following sections we present various aspects of NP architectures and programming models. We also discuss the overall NP connectivity and the role of its external interfaces which dictate how well it can be integrated with the other system components.

2. Network processor architecture

NPs use parallel processing to take advantage of the data parallelism present in packet streams. They employ multiple processing engines (PEs) to perform packet-processing tasks concurrently. Various aspects of NP architecture are discussed in the following sections.

2.1. Parallel and pipelined models

From the architecture and programming point of view, NP designs can be divided into two basic types: *Parallel* and *pipelined* as shown in Fig. 2. The PE in the parallel model is typically a scaled-down RISC-based architecture with some special bit-manipulation instructions suitable for packet processing. We refer to this type of PE as a general-purpose processing engine. PEs are generally simple and do not have complicated arithmetic, floating-point or numerous addressing modes. They typically have small instruction caches and also small data caches since most data is not re-used between packets. Many PEs can fit on a single chip since they are small. IBM's PowerNP NP is an example of the parallel model [8].

The *Task Scheduler* is responsible for assigning packets, as they arrive, to the PEs. The Task Scheduler is either hardwired, configurable or programmable. [1] It reads part of the packet header and dispatches it to a PE as soon as one becomes available. The Task Scheduler is also responsible for preserving packet sequence. Since packets belonging to the same flow may be dispatched to multiple PEs, the Task Scheduler puts them in the

[1] A processor-based Task Scheduler provides the ultimate flexibility since it is programmable.

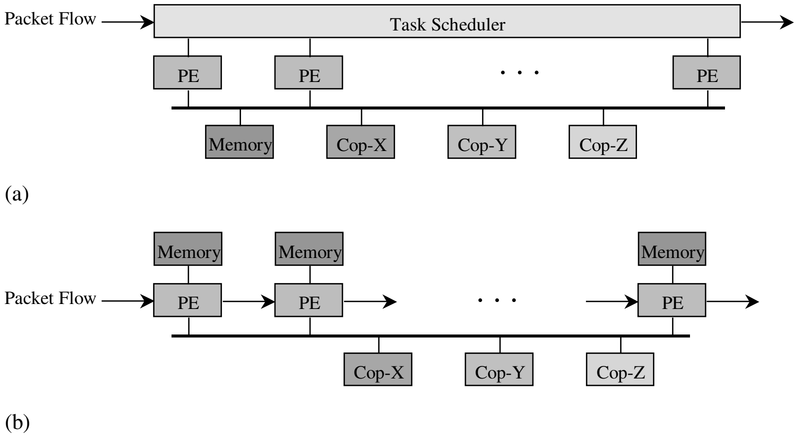


Fig. 2. Network processor base models: (a) parallel model, (b) pipelined model.

proper order for transmission once they are processed by PEs [6].

In the pipelined model the packet processing task is divided into multiple stages and each stage is designed to handle a certain category of task. A packet is passed from one stage to the next downstream stage as it is processed. Each pipeline stage employs one or multiple PEs that may be optimized for a specific task. We refer to this type of PE as a task-oriented processing engine. Each task-oriented PE is designed for a specific networking task and its instruction set is optimized for that task. For example, one set of PEs might be optimized to parse and classify the contents of packets. Another set might be optimized to search for matching the classification results with predefined values, and so forth. The Ezchip's NP-1 network processor is an example of the pipelined model [8].

The parallel and pipelined models are the same in terms of the total amount of processing that a packet can receive, however in the parallel model the processing budget for a PE is bigger and the throughput requirement is lower. For example, at 10 Gbps line speed with a 50-byte packet, a new packet arrives every 40 ns and there are 25 Mpps (million packets per second) to process. [2] Assume the parallel and pipelined model each has 16 PEs. In the pipelined model, at each stage a single PE can spend at most 40 ns processing each packet and the packet can be processed for at most $16 \times 40 = 640$ ns. Additionally, each PE must also have a minimum throughput of 25 Mpps. In the parallel model, a single PE can process a packet for a maximum of $16 \times 40 = 640$ ns and a PE must have a minimum throughput of $25/16 = 1,562,500$ pps.

The parallel and pipelined models can be used as the base to build different variations of these architectures such as the ones shown in Fig. 3. Such models can be designed to be more adaptable to varying application requirements. However these models typically make the programming task more challenging.

2.2. Memory organization

The NP memory holds three types of information: instruction code, control data, and packets. Each information type has different characteristics and performance requirements. The instruction code is stored in the *instruction memory*. The instruction code represents the application program and runs on the PEs. High-speed SRAMs with low-cycle access window are required for storing instructions. [3] The instruction memory subsystem

[2] The 25 Mpps figure is for illustration purposes. Network links are not utilized at 100%.

[3] With current technology, a low-cycle embedded SRAM can perform a read or write operation in 3 cycles.

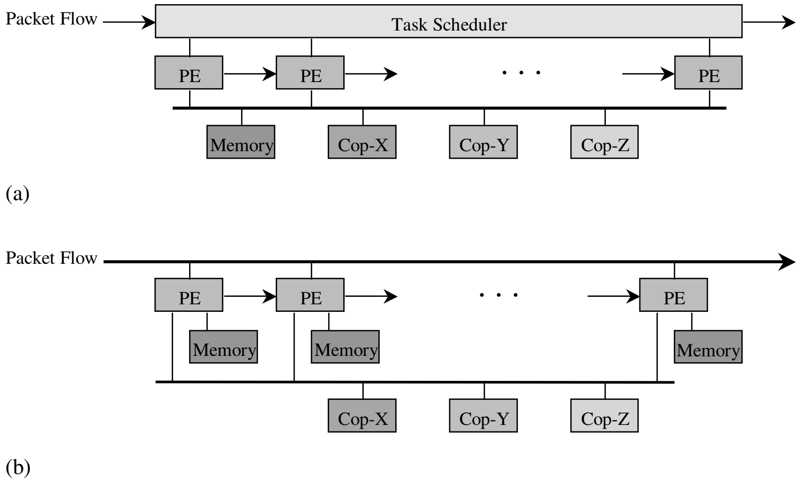


Fig. 3. Examples of variations in parallel and pipelined models: (a) parallel model with pipelined capability, (b) pipelined model with parallel bus.

needs to be structured in such a way to be able to feed all the PEs simultaneously. For NP hardware initialization and bring up, a PROM is also required to hold the boot code.

The instruction memory can be internal to the NP since it needs to hold a small amount of code. Typical low-layer protocol processing requires only on the order of a few kilobytes of code. However, for deep-packet processing involving higher-layer protocol translation and termination, several hundred kilobytes of instruction memory might be necessary.

Control data, including routing tables (e.g., IP addresses) and session contexts (e.g., TCP session information), is stored in the *control memory*. Control data needs to be stored in high-speed SRAMs. For protocol processing, several fields of an incoming packet might be used for searching in various tables containing policy, routing, and QoS information. For example, the table searches may involve: SA, DA, VLAN (of layer 2); IP address (of layer 3); TCP session (of layer 4); SSL context (of layer 5); URL (of layer 6) and so forth. The control memory bandwidth and its access time are critical factors in being able to sustain high-speed packet processing since several table access per packet are required. Storing tables in separate memory cores allows simultaneous table access to take place and yields better performance. Depending upon the type of protocol processing and the number of routing entries or session contexts to be supported, the control memory can range in size from a few hundred kilobytes to 10's of megabytes. For example, to support 50 K SSL-termination would require storing about 34 megabytes of context information (i.e., ~500 bytes per SSL context and ~180 bytes per TCP context).

Packets are stored in the *packet memory* during processing by the PEs. The packet memory subsystem design depends upon several factors including traffic characteristics (i.e., burst size, packet size, etc.), wire speed and the type of packet processing required (i.e., simple low-layer packet processing or complex deep-packet processing involving higher-layer protocols). For example, the packet memory requirement for an implementation involving only IP-packet forwarding is significantly different than another implementation involving TCP termination. The packet memory subsystem must provide sufficient bandwidth for multiple accesses per packet to achieve high-speed packet processing since each packet must be written to the packet memory and read back at least once. Additional packet memory accesses are required for reading packet data for processing and writing modified packet data before transmission. Large low-cost DRAMs used for packet memory in most todays NPs are not sufficient for high-speed deep-packet processing. Significantly faster memories with lower access times such as RLDRAMs or FCRAMs are better suited for high-speed NPs.

Memory is a major bottleneck for high-speed NPs—especially for deep-packet processing. Layer 4–7 protocol processing has significantly greater performance demands on memory than Layer 2–3 protocol processing. Higher-layer protocol processing involves reading and writing significantly more data from and to memory. As a result, the control and packet memories are heavily stressed.

The NP memory can be structured in three ways: *shared*, *distributed* and *hybrid* as shown in Fig. 4. The shared memory model suffers from limitations because of its lack of scalability and performance issues but it offers a simple programming model. The distributed memory model offers better scalability and increased performance but is more difficult to program. The hybrid model, a combination of shared and distributed models, offers better scalability and performance than the shared model and is simpler to program than the distributed model [10].

In the hybrid model, PEs are partitioned into a number of clusters each consisting of two or more PEs. Within a cluster, PEs have shared memory. PEs within a cluster may share instruction and packet memories. To avoid inter-cluster memory access for instruction code, the instruction code can be replicated in all clusters. This provides a single-image programming model since all the PEs execute the same code. The Task Scheduler dispatches all packets belonging to a flow to the same cluster to avoid inter-cluster memory access for packets.

PEs within a cluster can also share control memory since session context information is cluster-specific and need not be shared across clusters. This is because the Task Scheduler can dispatch all

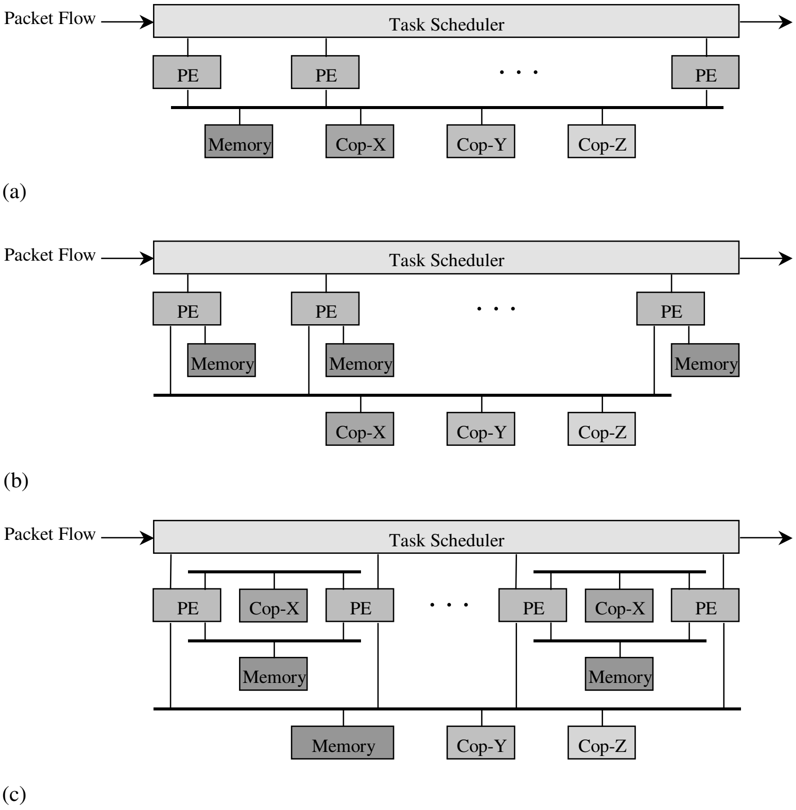


Fig. 4. Network processor memory models: (a) shared memory model, (b) distributed memory model, (c) hybrid memory model.

packets belonging to a flow to the same cluster. Route tables, however, need to be accessed by all PEs. More frequently used small-size tables can be replicated in the control memory of all clusters and large-size tables can be stored in a shared control memory accessible by all PEs.

2.3. Coprocessors

Hardwired or configurable coprocessors are employed in conjunction with PEs to accelerate common networking functions that are computationally intensive to perform in software. Coprocessors are typically shared among multiple PEs and they are accessed via message passing, memory map or special instructions. Typical functions performed by coprocessors include packet classification, table lookup and cryptographic functions.

A packet classification coprocessor parses a packet and compares preconfigured values with the values of the fields in the packet to determine if the packet includes any of the preconfigured values. It then generates some labels, based on the classification result, that are passed to a PE to assist in processing the packet.

A table lookup coprocessor assists in performing table searches. The basic function of a lookup coprocessor is to find the mapping value for a given key. Lookup coprocessors might be implemented using fixed-function CAMs or in algorithmic-dependent fashion based on hash tables or tree-structures. CAMs are ineffective for some entry types, have limited capacity and are expensive. Hash tables are suitable for fixed-length entries. Tree structures are suitable for variable-length entries such as the longest prefix match of IP addresses, URLs, etc. For example, in a tree-search coprocessor, tables might be represented as Patricia trees in which the termination of a search results in the address of a leaf page. Hash- and tree-based searches require multiple memory accesses.

Cryptographic functions are supported by crypto coprocessors to assist with security protocols such as IPSec and SSL. Crypto coprocessors support symmetric-key encryption algorithms (such as DES, TDES and AES), asymmetric-key encryption algorithms (such as RSA, DH and DSA), cryptographic hash algorithms (such as MD5 and SHA-1) and compression algorithms such as LZS.

A major aspect of defining an NP architecture is component placement. A coprocessor can be placed directly in the data-path, in a *streaming* mode, or in a *look-aside* mode as shown in Fig. 5. In the streaming mode, packets always pass through both the coprocessor and the NP complex. In the look-aside mode, the NP complex gets all the packets and only uses the coprocessor to assist with special-function processing. An advantage of the look-aside mode is that it can hide some latency, since it allows work on packets to take place in parallel by both PEs and coprocessors, but it makes system design more complex.

2.4. Multithreading

There are two main sources of latency on an NP: memory access (i.e., reads and writes) and coprocessor access (i.e., request to response time). Branch instructions can also introduce latencies but their latencies are typically shorter than memory or coprocessor access. Multithreading [4] provides a means for hiding latencies of various operations of an NP by allowing a PE to proceed with processing of alternate packets when processing of the current packet stalls for some reason such as a memory access. Multithreading can increase the utilization of PEs and can improve the overall performance of an NP.

There are different approaches to implementation of thread switching in PEs. In one approach the set of registers that are active at a thread switch point are saved in memory and are later restored when the execution returns to the thread again. Due to the save and restore memory operations, this approach can require many cycles to switch execution to another thread and in some scenarios it may even defeat the purpose of multithreading. In another approach a PE has multiple copies of its

[4] Throughout this paper we use the term multithreading to refer to multithreading for the sake of hiding latency, as opposed to the more general forms of software-based multithreading that are used in multi-tasking environments such as operating systems and servers.

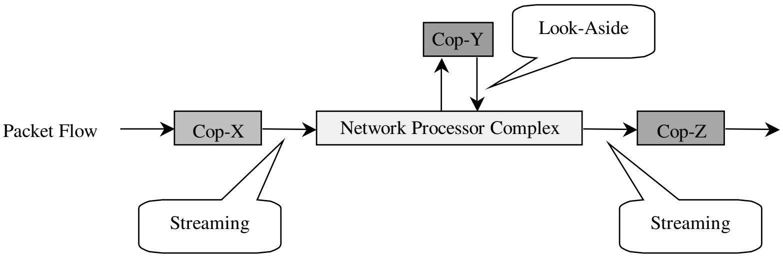


Fig. 5. Streaming and look-aside models.

local registers (e.g., one set of registers per thread). In this approach a single-stage PE can switch threads in one cycle since thread switching only requires pointing to another set of registers. In a multi-staged pipelined PE, additional cycles may be required since the pipeline may need to be cleared before thread switching can occur. Thread switching can be triggered by hardware automatically when execution stalls or by software using thread-switch instructions that are inserted in the code.

2.4.1. Multithreading analysis

In this section we develop an analytical model for multithreading analysis. Fig. 6 shows a multithreading example to illustrate our analysis model. We define the following parameters:

MemC The total *memory access cost* (i.e., read and write latencies) as represented by the code's instructions in terms of the number of memory cycles, where $MemC \geqslant 0$.

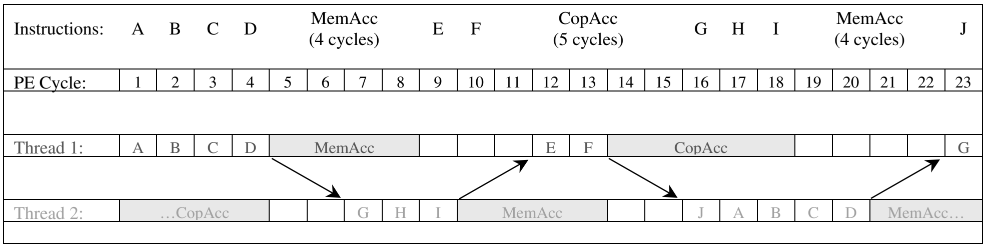
CopC The total *coprocessor access cost* (i.e., request to response latencies) as represented by the code's instructions in terms of the number of memory cycles, where $CopC \geqslant 0$.

L The total *number of latency sources* (i.e., memory accesses and coprocessor accesses) as represented by the code's instructions, where $L \geqslant 0$.

T The total *packet forwarding path length*, including execution of instructions, coprocessor operations, and memory accesses, in terms of the number of processor cycles, where $T \geqslant 1$.

n The *number of threads* that the processor supports, where $n \geqslant 1$.

C The *cost of thread switching* in terms of the number of processor cycles, where $(L \times C) \geqslant 0$. For example, in one implementation the processor may need to save and restore some registers or a multi-staged pipelined processor may require some extra cycles to clear its pipeline before it can switch to another thread.


Instructions: A B C D MemAcc (4 cycles) E F CopAcc (5 cycles) G H I MemAcc (4 cycles) J
PE Cycle: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Thread 1: A B C D MemAcc E F CopAcc G
Thread 2: …CopAcc G H I MemAcc J A B C D MemAcc…

$n = 2$ threads $C = 2$ cycles $MemC = 8$ cycles $CopC = 5$ cycles $T = 23$ cycles $L = 3$ $R = 1$

Fig. 6. Multithreading example for analytical model illustration.

R The *clock ratio* between the processor clock frequency and the memory clock frequency.

Given the above values, we calculate the probability, p, that the code is in "wait state" due to memory access or coprocessor access. [5] The total latency cost, K, in terms of the number of PE cycles is

$$K = R \times (MemC + CopC).$$

Thus, the wait-state probability is $p = K/T$.

The probability that every thread is in the wait state due to memory access or coprocessor access is then p^n. Note that in our calculation we assume threads are independent since packets arrive independently in time and as a result threads are being triggered independently. The probability that at least one thread is not in the wait state due to memory access or coprocessor access (i.e., it can be executed by the PE) is then $1 - p^n$.

From the above results, the probability that a given thread is switched can be represented by $p \times (1 - p^{n-1})$. This is basically the probability that the thread is in the wait state times the probability that there is at least one other thread not in the wait state which can be executed by the PE. Using this, we can then derive the probability, q, that the PE is in the "thread-switch state" as

$$q = \left(\frac{L \times C}{K}\right) \times p \times (1 - p^{n-1}) = \left(\frac{L \times C}{T}\right) \times (1 - p^{n-1}).$$

Now we can calculate the PE utilization, U, which represents the percentage of the time that the PE is doing useful work (i.e., executing the code and it is not in thread-switch or wait states). The PE utilization can be shown as

$$U = (1 - p^n) - \left(\frac{L \times C}{T}\right) \times (1 - p^{n-1}),$$

where $0 < U \leqslant 1$ and the valid range for the middle term is

$$0 \leqslant \frac{L \times C}{T} < \frac{1 - p^n}{1 - p^{n-1}}.$$

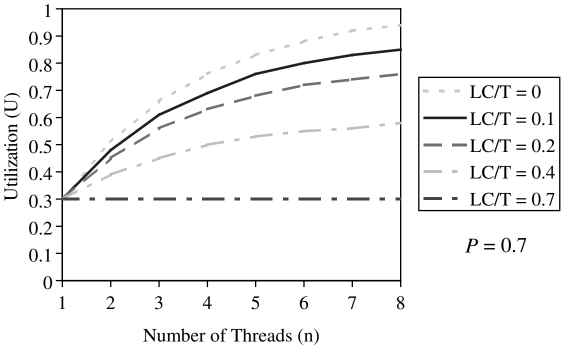


Fig. 7. Processing element utilization versus number of threads.

Fig. 7 shows the processing element utilization versus the number of threads supported for an example code path. From the figure and the utilization formula we can make the following observations:

- As the cost of thread switching, $L \times C$, approaches the total latency cost, K, the benefit of multithreading becomes less significant and in the limit case (i.e., $L \times C = K$) there is no benefit to multithreading, independent of the number of threads supported, since utilization is the same as a single-thread PE (i.e., $U = 1 - p$). This corresponds to the $(L \times C)/T = 0.7$ line in Fig. 7. Moreover, as the cost of thread switching, $L \times C$, exceeds the total latency cost, K, the utilization is negatively impacted by multithreading. That is, a single-thread PE performs better in this scenario.
- As the cost of thread switching, $L \times C$, compared to the code path length, T, decreases (i.e., as the code's path length increases) the benefit of low-cycle thread switching versus high-cycle thread switching becomes less significant and in the limiting case, that is $[(L \times C)/T] \rightarrow 0$, there is no benefit since utilization becomes the same as the ideal case (i.e., $U = 1 - p^n$).
- Independent of how many threads are supported, the utilization is limited by the cost

[5] Other sources of latency, such as branch instructions, can be easily included in the analysis but for simplicity of presentation they are not shown here.

of thread switching. In the limit case, that is as $n \to \infty$, the utilization becomes $U = 1 - [(L \times C)/T]$.

It should be noted that multithreading increases PE utilization but at the cost of introducing jitter, also known as delay variation, and longer delay in the packet stream. In a single-threaded PE, the PE is dedicated to processing a single packet at a time. So, it takes the same amount of time to process packets of the same type, however this is not the case with multithreading due to thread switching. In addition with multithreading, the packet stays in the system longer and as the result more system buffer space is required. These negative attributes of multithreading are not important for non-realtime data traffic but for realtime traffic (such as voice or video), which is sensitive to delay and jitter, these adverse effects are important. From this perspective, multithreading might be a selectable option that can be enabled or disabled depending upon the application via some instructions inserted in the code.

2.5. Support for traffic management

An NP may have specialized-hardware to assist software with traffic management functions or it may rely completely on software or an external device for traffic management. QoS guarantees for diversified user traffic, one of the most important issues for networking applications, can be enabled via traffic management [3]. QoS has four basic attributes that impact the user application: *availability*, *throughput*, *delay* (i.e., latency), and *jitter* (i.e., delay variation).

The percentage of time that the network is functional when the user application needs it is known as *availability*. High availability is achieved through a combination of reliable hardware and software components and redundancy. The average amount of user traffic delivered over a period of time is known as *throughput*. Throughput is typically measured in kilobits per second, megabits per second or gigabits per second. Intentional packet drops by the network or packet loss due to errors impacts the user-traffic throughput.

The average time that it takes for a user packet to travel from the ingress to the egress point of the network is known as *delay*. Realtime interactive applications such as voice and video are highly intolerant of delay. For more than 6 ms round-trip delay, echo cancellers for voice becomes necessary and when delay exceeds 200 ms, interactive voice becomes very cumbersome. *Delay variation*, the difference in delay experienced by packets belonging to the same traffic flow, causes choppiness in voice or video. Buffering can be used to overcome delay variation, but it introduces additional delay.

QoS guarantees for diversified user traffic can be achieved in two ways: (1) using overlay networks with bandwidth over-provisioning, or (2) using intelligent traffic management functions. Most service providers build their networks today using the first approach, in which they assign different types of traffic to different networks (e.g., voice and data) and provide sufficient additional bandwidth in the network to avoid the need for QoS mechanisms. This is, however, an expensive solution. The second approach involves using intelligent traffic management functions such as *classification*, *metering*, *marking*, *policing*, *scheduling*, and *shaping* [11,12,14]. For the latter approach, an NP may include specialized-units to support some of these functions in hardware (Fig. 8).

A packet *classifier* classifies packets based on the content of some portion of the packet header according to some specified rules and identifies packets that belong to the same traffic flow. A predefined *traffic profile* then specifies the properties (such as rate and burst size) of the selected traffic flow by the classifier. The traffic profile has preconfigured rules for determining whether a packet conforms to the profile. A traffic *meter* measures the rate of traffic flow, selected by the classifier, against the traffic profile to determine whether a particular packet is conforming to the profile. For example, a traffic profile based on a "token–bucket" scheme may define a token–bucket meter with token generation rate r and burst size b. In this example, a packet does not conforming to the profile if there are insufficient tokens available in the bucket when it arrives.

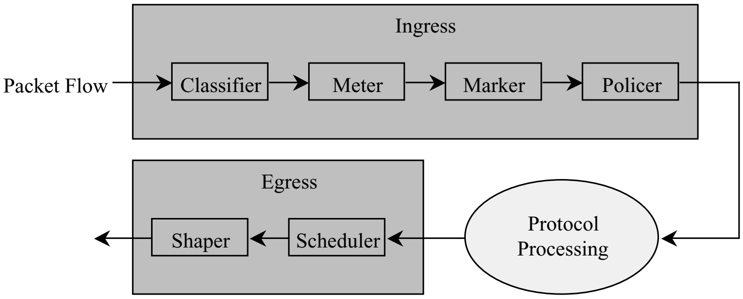


Fig. 8. Network processor support for traffic management.

A *marker* marks a packet based on the metering result. The marker has rules for marking packets based on whether they exceed the contracted traffic profile. There are several standards and proprietary schemes for packet marking. For example, Diffserv defines a single rate three color marker (srTCM) which marks IP packets either green, yellow or red according to three traffic parameters: committed information rate (CIR), committed burst size (CBS), and excess burst size (EBS). The srTCM-marking scheme marks an IP packet green if it does not exceed CBS, yellow if it exceeds CBS but not EBS, and red otherwise. These markings are then used to implement congestion control and queuing policies as established by the network administrator.

During periods of congestion, the *policer* discards packets entering the NP input queues. Packets are discarded in a fair fashion based on their marking (e.g., red packets first, yellow packets next and green packets last) when the input queues exceed some threshold values. As a result, fairness of resource usage is achieved since conforming-traffic is forwarded and non-conforming traffic is discarded. A congestion avoidance algorithm such as the random early detection (RED) or weighted random early detection (WRED) can be used for IP flows to control the average size of input queues. Such schemes start discarding packets from selected traffic flows when the input queues begin to exceed some threshold values.

Packet *scheduling* schemes are used to intelligently control how packets get serviced once they arrive at the NP output ports. To meet the various QoS requirements of a multi-service platform, a scheduler may support a mix of flexible queuing disciplines such as

- *Priority queuing* (PQ): Packets are assigned to different queues according to some priority scheme and the scheduler services the highest-priority queue first, the second highest-priority queue next and so forth. Empty queues are skipped.
- *Round robin queuing* (RRQ): Packets are assigned to various queues depending upon their types. Queues are then serviced by the scheduler one packet at a time in round-robin order, thereby providing a fair share of the output port to each queue.
- *Weighted fair queuing* (WFQ): This scheme is also known as class-based queuing (CBQ) and custom queuing (CQ). Packets are assigned into various queue classes and each queue is assigned a relative weight (e.g., 20% realtime, 30% interactive, and 50% file transfer). The scheduler services queues in a round-robin fashion in proportion to the weights assigned to queues.

Per-flow queuing with hierarchical scheduling is needed to provide proper QoS to each flow rather than a flow-class in an aggregated way. This is also known as hierarchical resource- or link-sharing scheduling [13]. Per-flow queuing with hierarchical scheduling enables traffic tracking and scheduling based on individual flows. This allows scheduling decisions to be based on, for example, application types and the associated subscribers and service providers, as shown in Fig. 9.

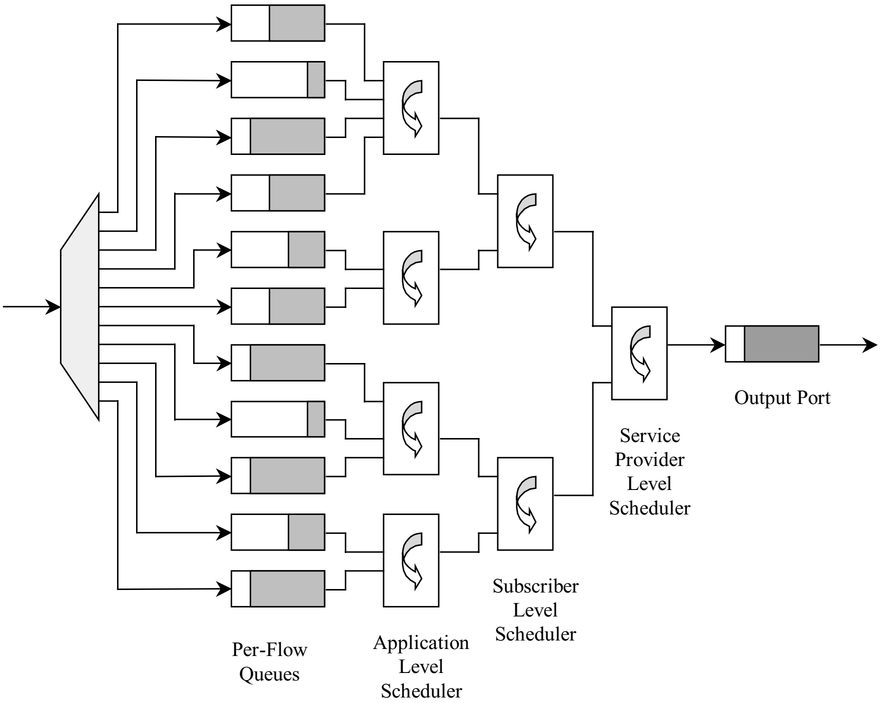

Output Port
Service Provider Level Scheduler
Subscriber Level Scheduler
Per-Flow Queues
Application Level Scheduler

Fig. 9. Per-flow queuing with 3-level hierarchical scheduling.

The *shaper* smoothes the traffic flow exiting the system in order to bring the traffic flow into compliance with the contracted traffic profile. This also allows the network administrator to control the traffic rate of an outbound interface. Shapers are typically implemented using a token–bucket scheme with a queue in which a packet is forwarded when there are sufficient tokens available in the bucket. When there are insufficient tokens available in the bucket, the packet is put on the queue and delayed until enough tokens are generated to service the queue.

2.6. External interfaces

An NP is typically a portion of a larger system and as the result its interfaces dictate how well it can be integrated with the other components of the system. The NP connectivity along with the overall flow of data throughout the system becomes a key factor in how data should flow through each subsystem component. Fig. 10 shows the basic external interfaces for an NP. Depending upon the targeted applications and environment, an NP may provide some or all of the interfaces shown in the figure.

The line interface is for attaching an NP to network ports through external framers and physical layer devices. An NP may include on-chip MACs to allow it to connect directly to external PHY devices. Typical line interfaces for NPs include: SMII for Fast Ethernet, GMII and/or TBI for Gigabit Ethernet, UTOPIA, POS, or the NPF [6]-adopted SPI-4.2. The line interface bandwidth determines how much network traffic can flow into or out of the NP.

The switch interface enables an NP to be attached to an external switch fabric. The switch interface must provide sufficient bandwidth to

[6] The Network Processing Forum (NPF) is an industry consortium which develops standards for NP hardware and software interfaces.

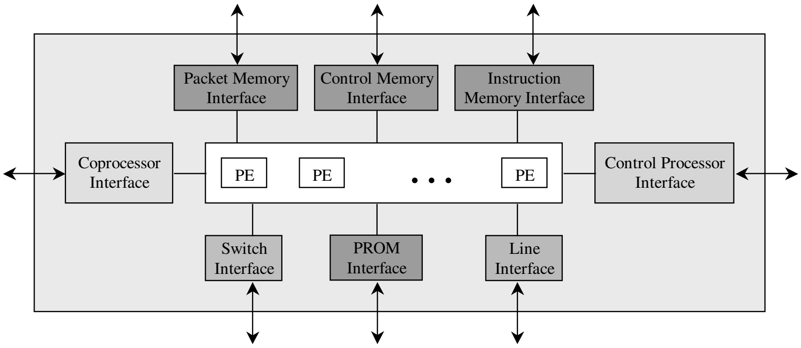


Fig. 10. Network processor main external interfaces.

allow all the line interface traffic to move across the switch interface. Overheads including switch fabric headers, NP headers, and in-band communication, needs to be accounted for in determining the over-speed required at the switch interface. Typically, 30–40% excess bandwidth on top of the line interface bandwidth is required. CSIX-L1 is the NPF standard for the switch fabric interface. Additionally, most switch fabric vendors have their own proprietary switch interfaces.

The memory interfaces are used for attachment to external packet, control and instruction memories, as discussed earlier. The multiplicity of memories along with their types and the bandwidth of these interfaces have major implications on the overall performance of an NP.

There are a wide variety of coprocessors that may be attached externally to an NP such as CAMs, lookup coprocessors and security coprocessors. Like the other interfaces, there is no single standard for the coprocessor interface. The NPF LookAside Phase 1 interface, which is based on QDR, may be used for CAM attachment. RapidIO or HyperTransport (also known as LDT) are more suitable for generic coprocessors (such as security coprocessors), since they provide support for memory coherency.

The control processor interface provides a means for the NP to communicate with an external control plane processor (i.e., host processor). Typical control plane interfaces include PCI, RapidIO and HyperTransport. PCI is a good choice, if the control plane traffic is expected to be low, since many GPPs support PCI. RapidIO or HyperTransport are more suitable for applications that require high data rates between the control processor and NP.

3. Network processor programming

The parallel architecture can provide a *Run-to-Completion* (RTC) programming model which offers the appearance of writing software for only one thread but results in the spreading of packets over many threads. This model allows the programmer to see a single thread which can access the entire instruction memory space and all the shared resources such as control memory, tables, counters, etc. The RTC model is based on the symmetric multiprocessor (SMP) architecture in which multiple PEs share the same memory. The PEs are used as a pool of processing resources, all executing simultaneously, either processing data or in idle mode waiting for work. The IBM's PowerNP network processor is an example of the RTC model.

In the pipelined architecture, which is based on some form of a distributed programming model, each pipeline PE is optimized to handle a certain category of tasks and instructions. In this model, the application program is partitioned among pipeline stages. A weakness in the pipeline model is the necessity of evenly distributing the work at each segment of the pipeline. When the work is not properly distributed, the flow of work through the pipeline is disrupted. For example, if one segment is over-allocated, then that segment of the pipeline will stall out preceding segments, and will starve out successive segments. The Intel's IXP1200 NP is an example of the pipelined model.

3.1. Programming challenges

With the high degree of performance expected of NPs, the trend is to provide operating system (OS) functionality in hardware and expose an interface to the application. Thus, there might be only a small software component to the OS or the OS might even be non-existent. For example, a hardware task scheduler can perform packet assignment to PEs, special hardware can handle the common I/O paths for packets, memory can be represented as free lists and so forth. These would avoid the need for separate OS service routines. However, the lack of a standard programming interface and an OS-level service make NP programming more complex than GPP [5].

To achieve good performance on NPs, application programs often need to be written in assembly or a combination of assembly and a high-level language (such as "C"). High-level language compilers for NPs are useful for initial software development but hand-optimization is often necessary for efficient final code [1]. This deficiency is mainly due to NP compliers' immaturity which should resolve over time as NP-specific code optimization techniques are developed for compilers.

The application software breakdown into control plane and data plane with one piece executing on a control-plane processor and the other on an NP requires a communication interface between the two. The application software needs to be "adapted" for each NP, since there is no standard for this interface today. The IETF's ForCES Working Group and NPF are in the process of developing standards for this interface.

4. Conclusion

NPs are becoming critical components of networking equipments to support evolving sophisticated applications at extraordinary performance levels. In this paper, we presented various NP architectures and analyzed their implications on the programming models. We examined the internal organizational structure and functions of the NP building blocks including processing engine, task scheduler, memory, coprocessor and traffic manager. We addressed the overall NP connectivity and the role of its external interfaces which dictate how well it can be integrated with the other system components. We discussed the NP programming models and the software development challenges presented by NPs.

The number of NPs with widely varied requirements and heterogeneous micro-architectures has grown at an astonishing rate. At the same, networking applications are constantly evolving and new application domains for NPs are still emerging. These factors make evaluation and comparison of NPs an increasingly challenging task. A systematic benchmarking methodology and evaluation scheme in terms of functionality and architectural aspects for the maturing NP field is needed to complement the approaches proposed in [2,7,9].

References

[1] J. Wagner, R. Leupers, C compiler design for a network processor, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20 (11) (2001) 1302–1308.

[2] M.A. Franklin, T. Wolf, A network processor performance and design model with benchmark parameterization, in: Proceedings of International Symposium on High-Performance Computer Architecture, February 2002, pp. 63–74.

[3] H. Shimonishi, T. Murase, A network processor architecture for flexible QoS control in very high speed line interfaces, in: Proceedings of IEEE Workshop on High Performance Switching and Routing, May 2001, pp. 402–406.

[4] T. Spalink, S. Karlin, L. Peterson, Y. Gottlieb, Building a robust software based router using network processors, Operating Systems Review 35 (5) (2001) 216–229.

[5] D. Herity, Network processor programming, Embedded Systems Programming 14 (8) (2001) 33–52.

[6] T. Wolf, M.A. Franklin, Locality aware predictive scheduling of network processors, in: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, November 2001, pp. 152–159.

[7] P. Crowley, M.E. Fiuczynski, J.L. Baer, B.N. Bershad, Characterizing processor architectures for programmable network interfaces, in: Proceedings of International Conference on Supercomputing, May 2000, pp. 54–65.

[8] L. Gwennap, B. Wheeler, A guide to network processors, Published by The Linley Group, November 2001.

[9] D. Suryanarayanan, G.T. Byrd, J. Marshall, A methodology and simulator for the study of network processors, in: Proceedings of International Symposium on High-Performance Computer Architecture, February 2002, pp. 18–29.

[10] J. Protic, M. Tomasevic, V. Milutinovic, Distributed shared memory: concepts and systems, IEEE Parallel and Distributed Technology 4 (2) (1996) 63–77.

[11] D. Chamberland, B. Sanso, Overall design of reliable IP networks with performance guarantees, in: Proceedings of IEEE International Conference on Communications, June 2000, pp. 1145–1151.

[12] S. Bakiras, V. Li, Efficient resource management for end to end QoS guarantees in DiffServ networks, in: Proceedings of IEEE International Conference on Communications, May 2002, pp. 1220–1224.

[13] S. Chen, K. Nahrstedt, Hierarchical scheduling for multiple classes of applications in connection-oriented integrated-service networks, in: Proceedings of International Conference on Multimedia Computing and Systems, 1999, pp. 153–158.

[14] S. Nelakuditi, S. Varadarajan, Z.L. Zhang, On localized control in QoS routing, IEEE Transactions on Automatic Control 47 (6) (2002) 1026–1032.

Mohammad Peyravian is a network processor architect at IBM Microelectronics in Research Triangle Park, North Carolina. He received a Ph.D. in Electrical Engineering from the Georgia Institute of Technology in 1992. His interests include networking, network processors, cryptography, and security. Mohammad has published over 40 journal and conference papers and has over 30 patents in networking and cryptography/security.

Jean Calvignac is an IBM Fellow at the IBM Microelectronics in Research Triangle Park, North Carolina. He initiated the IBM's network processor activities with his team in 1998. Previously, he was responsible for system design of the ATM switching products at the IBM La Gaude in France. Jean received his Engineering degree in 1969 from the Grenoble Polytechnique Institute, France. He is a Fellow member of the IEE (in Europe) and a senior member of IEEE.