12

Embedded Multi-Core Processing for Networking

Theofanis Orphanoudakis

University of Peloponnese
Tripoli, Greece
fanis@uop.gr

Stylianos Perissakis

Intracom Telecom
Athens, Greece
sper@intracom.gr

CONTENTS

12.1 Introduction

While advances in wire-line and wireless transmission systems have provided ample bandwidth surpassing customer demand at least for the near future, the bottleneck for high-speed networking and enhanced service provisioning has moved to processing. Network system vendors try to push processing at the network edges employing various techniques. Nevertheless, the networking functionality is always proliferating as more and more intelligence (such as multimedia content delivery, security applications and quality of service (QoS)- aware networks) is demanded. The future Internet is expected to provide a data-centric networking platform providing services beyond today's expectations for shared workspaces, distributed data storage, cloud and grid-computing, broadcasting and multi-party real-time media-rich communications and many types of e-services such as sophisticated machine-machine interaction between robots, e-health, and interactive e-learning. Thus, the model of routing/switching devices has been augmented to enable the introduction of value added services involving complex network processing over multiple protocol stacks and the raw data forwarding functionality has been left only as the major task of large core switches that are exclusively assigned with this task. To cope with this demand, system designers have leaned on micro-electronic technology to embed network processing functions in either fixed or programmable hardware as much as possible. This led to a new generation of multi-core embedded systems specifically designed to tackle network processing application requirements.

In the past the power required for the processing of protocol functions at wire speed was usually obtained either by generic microprocessors (also referred to as central processing units, CPUs) designed with the flexibility to perform a variety of functions, but at a slower speed, or application specific integrated circuits (ASICs) designed to meet a specific functional requirement with high efficiency. Notwithstanding the requirement for high capacity and high quality of the offered services, the development cost of such systems (affected by the system component cost, application development cost, time-to-market as well as time-in-market) remains a critical factor in the development of such platforms. Hybrid programmable system-on-chip (SoC) devices integrating either generalized or task-specific processing cores called in general network processing units (NPUs) have recently deposed generic CPU-based products from many networking applications, extending the scalability (i.e., time-in-market) and performance of these products, therefore reducing cost and maximizing profits. In general NPUs can be defined as programmable embedded multi-core semiconductor systems optimized for performing wire speed operations on packet data units (PDUs). The development of such complex devices with embedded CPUs and diverse IP blocks has introduced a new paradigm in micro-electronics design as well in exploitation, programming and application porting on such devices.

The requirements of applications built on NPU-based devices are expanding dramatically. They must accommodate the highest bit rate on the one hand while coping with protocol processing of increased complexity on the other. In general the functionality of these protocols that spans the area of network processing can be classified as shown in Figure 12.1.

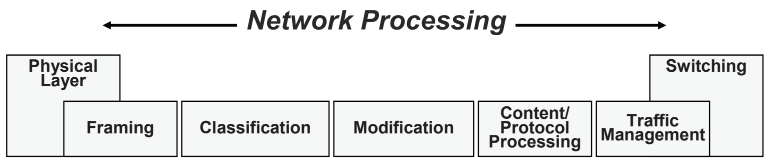

FIGURE 12.1: Taxonomy of network processing functions.

Physical layer processing and traffic switching are mostly related to the physical characteristics of the communications channel and the technology of the switching element used to interconnect multiple network nodes. The physical layer processing broadly includes all functions related to the conversion from transport media signals to bits. These functions include reception of electronic/photonic/RF signals and are generally classified in the different sub-layers such as the physical medium dependent (PMD), physical medium attachment (PMA) and physical coding sub-layer (PCS) resulting in appropriate signal reception, demodulation, amplification and noise compression, clock recovery, phase alignment, bit/byte synchronization and line coding. Switching includes the transport of PDUs from ingress to egress ports based on classification/routing criteria. For low rate applications switching is usually implemented through shared memory architectures, whereas for high rate applications through crossbar, bus, ring or broadcast and select architectures (the latter especially applied in the case of optical fabrics). The most demanding line-rates that motivated the wider introduction of NPUs in networking systems range in the order of 2.5 to 40 Gbps (OC-48, OC-192 and OC-768 data rates of the synchronous optical networking standard: SONET).

Framing and deframing includes the conversion from bits to PDUs, grouping bits into logical units. The protocols used are classified as Layer 2 protocols (data link layer of the OSI reference architecture) and the logical units are defined as frames, cells or packets. PDU conversion may also be required in the form of segmentation and reassembly. Most usually some form of verification of the PDU contents also needs to be applied to check for bit and field errors requiring the generation/calculation of checksums. In the more general case the same functionality is extended to all layers of the protocol stack since all telecommunication protocols employ some packetization and encapsulation techniques that require the implementation of programmable field extraction and modification, error correction coding and segmentation and reassembly (including buffering and memory management) schemes.

Classification includes the identification of the PDUs based on pattern matching to perform field lookups or policy criteria, also called rules. Many protocols require the differentiation of packets based on priorities, indicated in bits, header fields or multi-field (layers 2 up to 7 information fields) conditions. Based on the parsing (extraction of bits/fields) of the PDUs, pattern matching in large databases (including information about addresses, ports, flow tables etc.) is performed. Modification facilitates actions on PDUs based on classification results. These functions perform marking/editing of PDU bits to implement network address translation, ToS (type of service), CoS (class of service) marking, encapsulation, recalculation of checksums etc.

Content/protocol processing (i.e., processing of the entire PDU payload) may be required in case of compression (in order to reduce bandwidth load through the elimination of data redundancy) and encryption (in order to protect the PDU through scrambling, using public/private keys etc.) as well as deep packet inspection (DPI) for application aware filtering, content based routing and other similar applications. Associated functions required in most cases of protocol processing include the implementation of memory management techniques for the maintenance of packet queues, management of timers and implementation of finite state machines (FSMs).

Traffic engineering facilitates differentiated handling for flows of PDUs characterized by the same ToS or CoS, in order to meet a contracted level of QoS. This requires the implementation of multiple queues per port, loaded based on classification results (overflow conditions requiring additional intelligent policing and packet discard algorithms) and served based on specific scheduling algorithms.

In the packet network world, the CPU traditionally assumes the role of packet processor. Many protocols for data networks have been developed with CPU-centered architectures in mind. As a result, there are protocols with variable length headers, checksums in arbitrary locations and fields using arbitrary alignments. Two major factors drive the need for NPUs: i) increasing network bit rates, and ii) more sophisticated protocols for implementing multi-service packet-switched networks. NPUs have to address the above communications system performance issues coping with three major performance-related resources in a typical data communication system:

1. Processing cores
2. System bus(es)
3. Memory

These requirements drive the need for multi-core embedded systems specifically designed to alleviate the above bottlenecks by assigning hardware resources to efficiently perform specific network processing tasks. NPUs mainly aim to reduce CPU involvement in the above packet processing steps, which represent more or less independent functional blocks and generally result in the high-level specification of an NPU as a multi-core system.

12.2 Overview of Proposed NPU Architectures

12.2.1 Multi-Core Embedded Systems for Multi-Service Broadband Access and Multimedia Home Networks

The low-cost, limited-performance, feature-rich range of multi-core NPUs can be found in market applications that are motivated by the trend for multi-service broadband access and multimedia home networks. The networking devices that are designed to deliver such kind of applications to the end users over a large mixture of networking technologies and a multitude of interfaces face stringent requirements for size, power and cost reduction. These requirements can only be met by a higher degree of integration without sacrificing though performance and the flexibility to develop new applications on the same hardware platform over time. Broadband access networks use a variety of access technologies, which offer sufficient network capacity to support high-speed networking and a wide range of services. Increased link capacities have created new requirements for processing capabilities at both the network and the user premises.

The complex broadband access environment requires inter-working devices connecting network domains to provide the bridge/gateway functionality and to efficiently route traffic between networks of diverse requirements and operational conditions. These gateways constitute the enabling technology for multimedia content to reach the end users, advanced services to be feasible, and broadband access networking to become technically feasible and economically viable. A large market share of these devices includes the field of home networks, including specialized products to interconnect different home appliances, such as PCs, printers, DVD players, TV, over a home network structure, letting them share broadband connections, while performing protocol translation (e.g., IP over ATM) and routing, enforcing security policies etc. The need for such functionality has created the need for a new device, the residential gateway (RG).

The RG allows consumers to network their PCs, so they can share Internet access, printers, and other peripherals. Furthermore, the gateway allows people to play multiplayer games or distribute movies and music throughout the home or outdoors, using the broadband connection. The RG also enables interconnection and interworking of different telephone systems and services, wired, wireless, analog and IP-based, and supports telemetry and control applications, including lighting control, security and alarm, and in-home communication between appliances [44].

A set of the residential gateway functions includes carrying and routing data and voice securely between wide area network (WAN) and local area network (LAN), routing data between LANs, ensuring only the correct data is allowed in and out of the premises, converting protocols and data, selecting channels for bandwidth-limited LANs, etc. [20]. RGs with minimal functionality can be transparent to multimedia applications (with the exception of the requirement for QoS support for different multimedia traffic classes). However, sophisticated RGs will be required to perform media adaptations (i.e., POTS to voice over IP VoIP) or stream processing (i.e., MPEG-4) as well as control functions to support advanced services like for example stateful inspection firewalls and media gateways. All of the above networking applications are based on a protocol stack implementation involving processing of several layers and protocols. The partitioning of these functions into system components is determined by the expected performance of the overall system architecture. Recent trends employ hardware peripherals as a means to achieve acceleration of critical time-consuming functions. In any case the implementation of interworking functions is mainly performed in software. It is evident though that software implementations fail to provide real-time response, a feature especially crucial for voice services and multimedia applications.

To better understand the system level limitations of a gateway supporting these kinds of applications for a large number of flows, we show in Figure 12.2 the available system/processor clock cycles per packet, for different clock frequencies, for three different link rates. Even in the best case where one processor instruction could be executed in each cycle (which is far from true due to pipeline dependencies, cache misses etc.), the number of instructions that can be executed per packet is extremely low compared to the required processing capacity of complex applications. Taking into account also the memory bottlenecks of legacy architectures, it is evident that the overall system level architecture must be optimized with respect to network processing in order to cope with demanding services and multimedia applications.

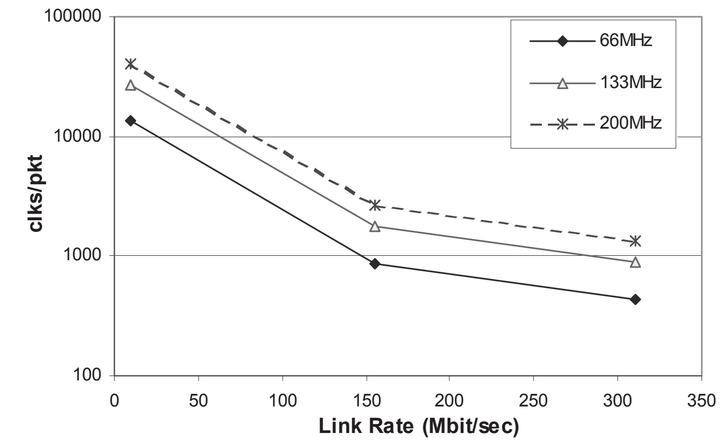

FIGURE 12.2: Available clock cycles for processing each packet as a function of clock frequency and link rate in average case (mean packet size of 256 bytes is assumed).

12.2.2 SoC Integration of Network Components and Examples of Commercial Access NPUs

Currently, the major trend in network processing architectures centers on their implementation by integrating multiple cores resulting in a system-on-chip (SoC) technology. SoC technology provides high integration of processors, memory blocks and algorithm-specific modules. It enables low cost implementation and can accommodate a wide range of computation speeds. Moreover, it offers a supporting environment for high-speed processor interconnection, while input/output (I/O) speed and the remaining off-SoC system speed can be low. The resulting architecture can be used for efficient mapping of a variety of protocols and/or applications. Special attention is focused on the edge, access, and enterprise markets, due to the scales of the economy in these markets. In order to complete broadband access deployment, major efforts are required to transfer the acquired technological know-how from the high-speed switching systems to the edge and access domain by either developing chips geared for the core of telecom networks that are able to morph themselves into access players, or by developing new SoC architectures tailored for the access and residential system market.

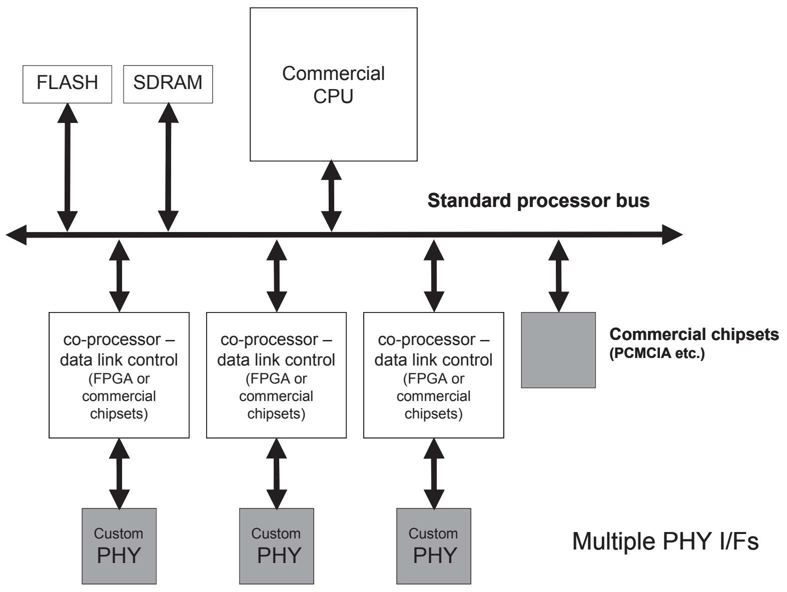
FLASH
SDRAM
Commercial CPU
Standard processor bus
co-processor – data link control (FPGA or commercial chipsets)
co-processor – data link control (FPGA or commercial chipsets)
co-processor – data link control (FPGA or commercial chipsets)
Commercial chipsets (PCMCIA etc.)
Custom PHY
Custom PHY
Custom PHY
Multiple PHY I/Fs

FIGURE 12.3: Typical architecture of integrated access devices (IADs) based on discrete components.

A common trend for developing gateway platforms to support multi-protocol and multi-service functionality in edge devices was until recently to use as main processing resources those of a commercial processor (Figure 12.3). Network interfaces were implemented as specialized H/W peripherals. Protocol processing was achieved by software implementations developed on some type of standard operating system and development platform. The main bottleneck in this architecture is apparently on one hand the memory bandwidth (due to the limited throughput of the main system memory) and on the other hand the limited speed of processing in S/W.

Driven by the conflicting requirements of higher processing power versus cost reduction, SoC architectures with embedded processor cores and increased functionality/complexity have appeared, replacing discrete component integrated access devices (IADs). Recent efforts to leverage NPUs in access systems aim to reduce the bottleneck of the central (CPU) memory. Furthermore, the single on-chip bus that interconnects all major components in typical architectures is another potential bottleneck. In an NPU-based architecture the bandwidth demands on this bus are reduced, because this bus can become arbitrarily wide (Figure 12.4) or alternatively the processor and peripheral buses can be separated. Therefore, such architectures are expected to scale better, being able to support network devices with higher throughput and more complex protocol processing than current gateways.

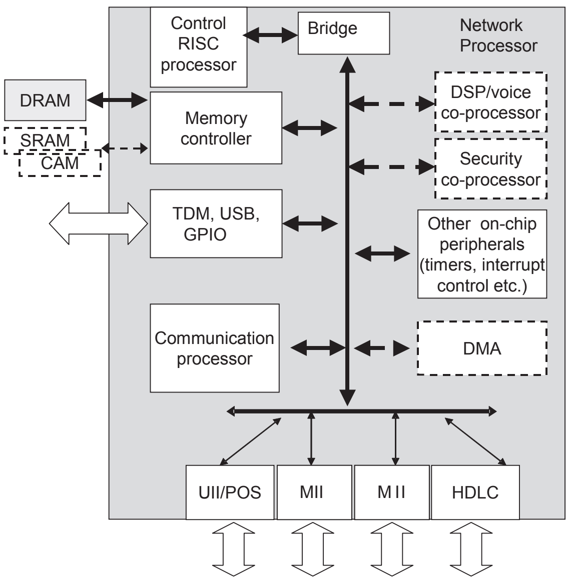

FIGURE 12.4: Typical architecture of SoC integrated network processor for access devices and residential gateways.

12.2.3 NPU Architectures for Core Network Nodes and High-Speed Networking and Switching

Beyond broadband access, the requirements for specialized multi-core embedded systems to perform network processing, as mentioned in the introduction of this section, have initially been considered in the context of replacing the high-performance but with limited programmability ASICs traditionally been developed to implement high-speed networking and switching in core network nodes. Core network nodes include IP routers, layer 3 fast, gigabit and 10 gigabit Ethernet switches, ATM switches and VoIP gateways. Next-generation embedded systems require a silicon solution that can handle the ever-increasing speed, bandwidth, and processing requirements. State-of-the-art systems need to process information implementing complex protocols and priorities at wire-speed and handle the constantly changing traffic capacity of the network. NPUs have emerged as the promising solution to deliver high capacity switching nodes with the required functionality to support the emerging service and application requirements. NPUs are usually placed on the data path between the physical layer and backplane within layer 3 switches or routers implementing the core functionality of the router and perform all the network traffic processing. NPUs must be able to support large bandwidth connections, multiple protocols, and advanced features without becoming a performance bottleneck. That is, NPUs must be able to provide wire-speed, non-blocking performance regardless of the size of the links, protocols and features enabled per router or switch port.

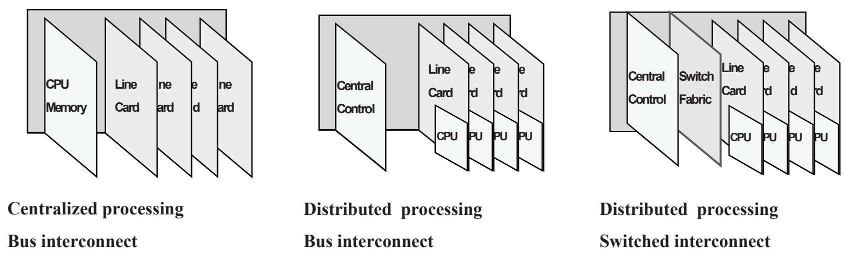

FIGURE 12.5: Evolution of switch node architectures: (a) 1st generation (b) 2nd generation (c) 3rd generation.

In the evolution of switching architectures, 1st and 2nd generation switches relied on centralized processing and bus interconnection-based architectures limiting local per port processing merely on physical layer adaptation. From the single CPU multiple line cards with single electrical backplane of 1st generation switches, technology advanced to distributed processing in its 2nd generation with one CPU per line card and a central controller for routing protocols and system control and management. A major breakthrough was the introduction of the switch fabric for inter-connection in the 3rd generation switches, to overcome the interconnection bandwidth problem, whereas the processing bottleneck was still treated with the same distributed architecture (Figure 12.5).

The PDU flow is shown in more detail in Figure 12.6. For the 1st generation switches shown in Figure 12.5 above, the network interface card (NIC) passes all data to CPU, which does all the processing, resulting in inexpensive NICs and overloaded interconnects (buses) and CPUs. The 2nd generation switches relieve the CPU overload by distributed processing, placing dedicated CPUs in each NIC; the interconnect bottleneck though remained. Finally 3rd generation switches introduced the switching fabric for efficient board-to-board communication over electronic backplanes.

NPUs mainly aim to reduce CPU involvement, used either in a centralized or distributed fashion and have been introducing the modifications to the architecture of Figure 12.6 as shown in Figure 12.7 below. In the centralized architecture (Figure 12.7a), the NIC passes all data to a high bandwidth NPU, which does all packet processing assuming the same protocol stack for all ports. Performance degrades with increased protocol complexity and increased numbers of ports. In a distributed architecture (Figure 12.7b) the CPU config-

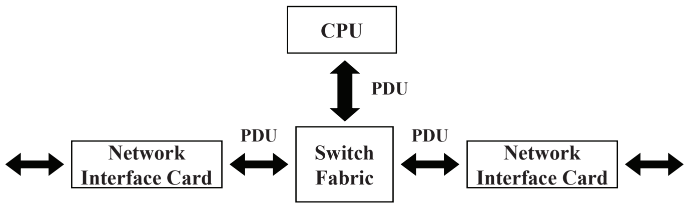

FIGURE 12.6: PDU flow in a distributed switching node architecture.

ures NPU execution and NPUs do all packet processing, possibly assisted by specialized traffic managers (TMs) for performing complex scheduling/shaping and buffer management algorithms. Each port can execute independent protocols and policies through a programmable NIC architecture.

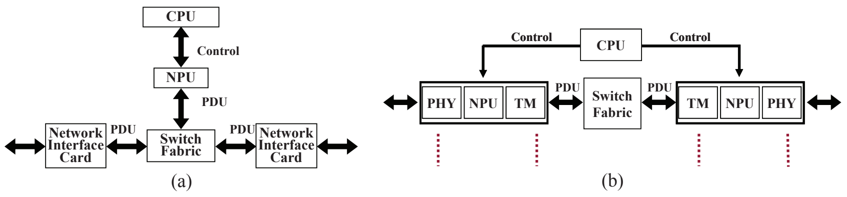

FIGURE 12.7: Centralized (a) and distributed (b) NPU-based switch architectures.

NPUs present a close coupling of link-layer interfaces with the processing engine, minimizing the overhead typically introduced in generic microprocessor-based architectures by device drivers. NPUs use multiple execution engines, each of which can be a processor core usually exploiting multi-threading and/or pipelining to hide DRAM latency and increase the overall computing power. NPUs may also contain hardware support for hashing, CRC calculation, etc., not found in typical microprocessors. Figure 12.8 shows a generic NPU architecture, which can be mapped to many of the NPUs discussed in the literature and throughout this chapter. Additional storage is also present in the form of SRAM (synchronous random access memory) and DRAM (dynamic random access memory) to store program data and network traffic. In general, processing engines are intended to carry out data-plane functions. Control-plane functions could be implemented in a co-processor, or a host processor.

An NPU's operation can be explained in terms of a representative application like IP forwarding, which could be tentatively executed through the following steps:

1. A thread on one of the processing engines handles new packets that arrive in the receive buffer of one of the input ports.

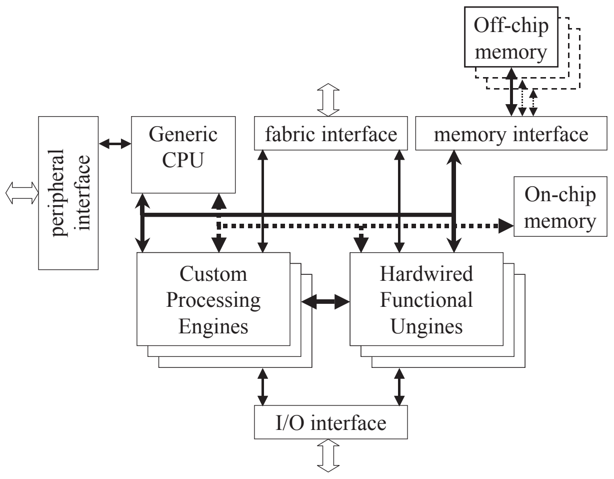

FIGURE 12.8: Generic NPU architecture.

2. The (same or alternative) thread reads the packet's header into its registers.

3. Based on the header fields, the thread looks up a forwarding table to determine to which output queue the packet must go. Forwarding tables are organized carefully for fast lookup and are typically stored in the high-speed SRAM.

4. The thread moves the rest of the packet from the input interface to packet buffer. It also writes a modified packet header in the buffer.

5. A descriptor to the packet is placed in the target output queue, which is another data structure stored in SRAM.

6. One or more threads monitor the output ports and examine the output queues. When a packet is scheduled to be sent out, a thread transfers it from the packet buffer to the port's transmit buffer.

The majority of the commercial NPUs fall mainly into two categories: The ones that use a large number of simple RISC (reduced instruction set computer) CPUs and those with a number (variable depending on their custom architecture) of high-end, special-purpose processors that are optimized for the processing of network streams. All network processors are system-on-chip (SoC) designs that combine processors, memory, specialized logic, and I/O on a single chip. The processing engines in these network processors are typically RISC cores, which are sometimes augmented by specialized instructions, multi-threading, or zero-overhead context switching mechanisms. The on-chip memory of these processors is in the range of 100KB to 1MB.

Within the first category we find:

- Intel IXP1200 [28] with six processing engines, one control processor, 200 MHz clock rate, 0.8-GB/s DRAM bandwidth, 2.6-Gb/s supported line speed, four threads per processor
- Intel IXP2400 and Intel IXP2800 [19] with 8 or 16 micro- engines, one control processor and 600 MHz or 1.6GHz clock rates, while also supporting 8 threads per processor
- Freescale (formerly Motorola) C-5 [6] with 16 processing units, one control processor, 200 MHz clock rate 1.6-GB/s DRAM bandwidth, 5-Gb/s supported line speed and four threads per processor
- CISCOs Toaster family [7] with 16 simple microcontrollers

All these designs generally adopt the parallel RISC NPU architecture employing multiple RISCs augmented in many cases with datapath co-processors (Figure 12.9(a)). Additionally they employ shared engines capable of delivering (N $\times$ port BW) throughput interconnected over an internal shared bus of 4 $\times$ total aggregate bandwidth capacity (to allow for at least two read/write operations per packet) as well as auxiliary external buses for implementing insert/extract interfaces to external controllers and control plane engines.

Although the above designs can sustain network processing from 2.5 to 10 Gbps, the actual processing speed depends heavily on the kind of application and for complex applications it degrades rapidly. Further, they represent a brute-force approach, in the sense that they use a large number of processing cores, in order to achieve the desired performance.

The second category includes NPUs like:

- EZChips NP1 [9] with a 240 MHz system clock that employs multiple specific-purpose (i.e., lookup) processors as shared resources without being tied to a physical port
- HiFns (formerly IBMs) PowerNP [17] with 16 processing units (picoprocessors), one control processor, 133 MHz clock rate, 1.6-GB/s DRAM bandwidth, eight-Gb/s line speed and two threads per processor, as well as specialized engines for look-up, scheduling and queue management

These designs may follow different approaches most usually found as either pipelined RISC architectures including specialized datapath RISC engines for executing traffic management and switching functions (Figure 12.9(a)), or generally programmable state machines which directly implement the required functions (Figure 12.9(b)). Both these approaches have the feature that the internal data path bus is required to offer only 1 $\times$ total aggregate bandwidth.

Although, the aforementioned NPUs are capable of providing a higher processing power for complicated network protocols, they lack the parallelism of the first category. Therefore, their performance, in terms of bandwidth

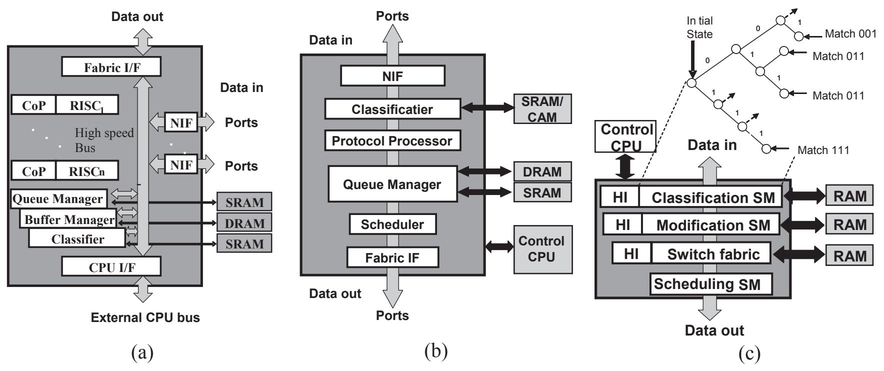

FIGURE 12.9: (a) Parallel RISC NPU architecture (b) pipelined RISC NPU architecture (c) state-machine NPU architecture.

serviced, is lower than the one of the first category whenever there is a large number of independent flows that should be processed.

Several of these architectures are examined in the next section, while the micro-architectures of several of the most commonly found co-processors and hardwired engines are discussed throughout this chapter.

12.3 Programmable Packet Processing Engines

NPUs are typical domain-specific architectures: in contrast to general purpose computing, their applications fall in a relatively narrow domain, with certain common characteristics that drive several architectural choices. A typical network processing application consists of a well-defined pipeline of sequential tasks, such as: decapsulation, classification, queueing, modification, etc. Each task may be of small to modest complexity, but has to be performed with a very high throughput, or repetition rate, over a series of data (packets), that most often are independent from each other. This independence arises from the fact that in most settings the packets entering a router, switch, or other network equipment, belong to several different flows. In terms of architectural choices, these characteristics suggest that emphasis must be placed on throughput, rather than latency. This means that rather than architecting a single processing core with very high performance, it is often more efficient to utilize several simpler cores, each one with moderate performance, but with a high overall throughput. The latency of each individual task, executed for each individual packet, is not that critical, since there are usually many independent data streams processed in parallel. If and when one task stalls, most of the time there will be another one ready to utilize the processing cycles made available. In other words, network processing applications are usually latency tolerant.

The above considerations give rise to two architectural trends that are common among network processor architectures: *multi-core parallelism*, and *multi-threading*.

12.3.1 Parallelism

The classic trade-off in computer architecture, that of performance versus cost (silicon area) manifests itself here as single processing engine (PE) performance versus the number of PEs that can fit on-chip. In application domains where there is not much inherent parallelism and more than a single PE cannot be well utilized, high single-PE performance is the only option. But where parallelism is available, as is the case with network processing, the trade-off usually works out in favor of many simple PEs. An added benefit of the simple processing core approach is that typically higher clock rates can be achieved. For these reasons, virtually all high-end network processor architectures rely on multiple PEs of low to moderate complexity to achieve the high throughput requirements common in the OC-48 and OC-192 design points. As one might expect, there is no obvious "sweet spot" in the trade-off between PE complexity and parallelism, so a range of architectures have been used in the industry.

Typical of one end of the spectrum are Freescale's C-port and Intel's IXP families of network processors (Figure 12.10). The Intel IXP 2800 [2][30] is based on 16 microengines, each of which implements a basic RISC instruction set with a few special instructions, contains a large number of registers, and runs at a clock rate of 1.4 GHz. The Freescale C-5e [30] contains 16 RISC engines that implement a subset of the MIPS ISA in addition to 32 custom VLIW processing cores (Serial Data Processors, or SDPs) optimized for bit and byte processing. Each RISC engine is associated with one SDP for the ingress path, that performs mainly packet decapsulation and header parsing, and one SDP for the egress path, that performs the opposite functions — those of packet composition and encapsulation.

Further reduction in PE complexity, with commensurate increase in PE count, is seen in the architecture of the iFlow Packet Processor (iPP) [30] by Silicon Access Networks. The iPP is based on an array of 32 simple processing elements called *Atoms*. Each Atom is a reduced RISC processor, with an instruction set of only 47 instructions. It is interesting to note, however, that many of these are custom instructions for network processing applications.

As a more radical case, we can consider the PRO3 processor [37]: its main processing engine, the reprogrammable pipeline module (RPM) [45] consists of a series of three programmable components: a field extraction engine (FEX), the packet processing engine proper (PPE), and a field modification engine (FMO), as shown in Figure 12.11. The allocation of tasks is quite straightforward: packet verification and header parsing are performed by FEX, general

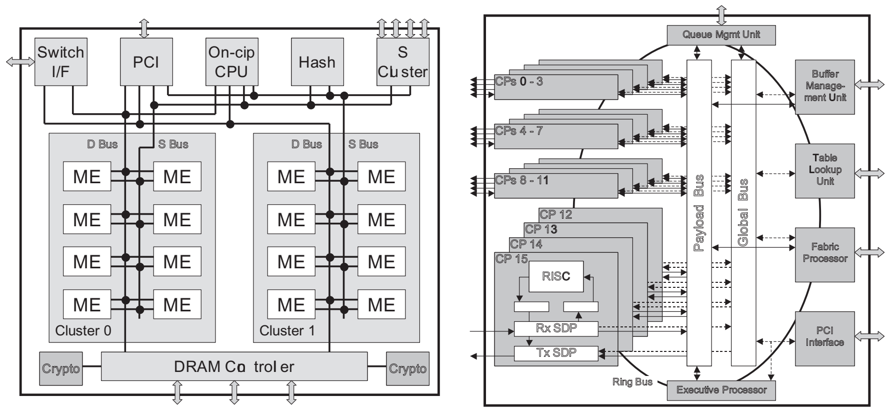

FIGURE 12.10: (a) Intel IXP 2800 NPU, (b) Freescale C-5e NPU.

processing on the PPE, and modification of header fields or composition of new packet headers is executed on the FMO. The PPE is based on a Hyperstone RISC CPU, with certain modifications to allow fast register and memory access (to be discussed in detail later). The FEX and FMO engines are bare-bones RISC-like processors, with only 13 and 22 instructions (FEX and FMO, respectively).

In another approach, a number of NPU architectures attempt to take advantage of parallelism at a smaller scale within each individual PE. *Instruction-level parallelism* is usually exploited by superscalar or Very-Long-Instruction-Word (VLIW) architectures. Noteworthy is EZchip's architecture [9][30], based on superscalar processing cores, that EZchip claims are up to 10 times faster on network processing tasks than common RISC processors. SiByte also promoted the use of multiple on-chip four-way superscalar processors, in an architecture complete with two-level cache hierarchy. Such architectures of course are quite expensive in terms of silicon area, and therefore only a relatively small number of PEs can be integrated on-chip. Compared to superscalar technology, VLIW is a lot more area-efficient, since it moves a lot of the instruction scheduling complexity from the hardware to the compiler. Characteristic of this approach are Motorola's SDP processors, mentioned earlier, 32 of which can be accommodated on-chip, along with all the other functional units.

Another distinguishing feature between architectures based on parallel PEs is the *degree of homogeneity*: whether all available PEs are identical, or whether they are specialized for specific tasks. To a greater or lesser degree, all architectures include special-purpose units for some functions, either fixed logic or programmable. The topic of subsequent sections of this chapter is to analyze the architectures of the more commonly encountered special-purpose units. At this point, it is sufficient to note that some of the known archi-

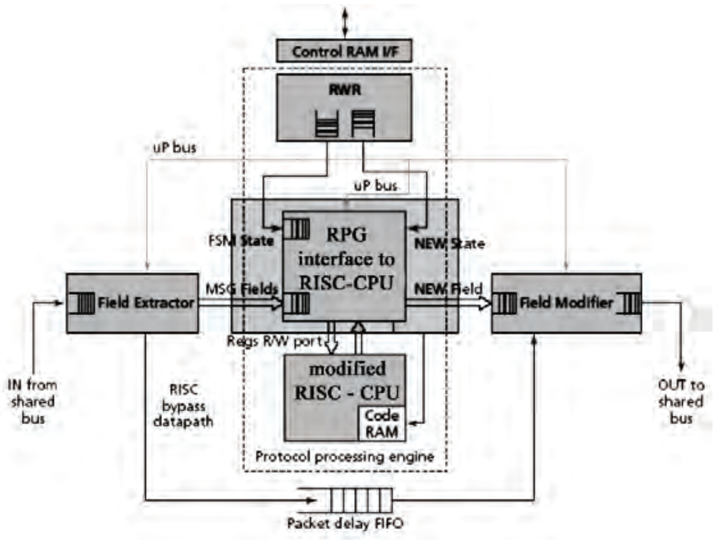

FIGURE 12.11: Architecture of PRO3 reprogrammable pipeline module (RPM).

tectures place emphasis on many identical programmable PEs, while others employ PEs with different variants of the instruction set and combinations of functional units tailored to different parts of the expected packet processing flow.

Typical of the specialization approach is the EZchip architecture: it employs four different kinds of PEs, or Task-OPtimized cores (TOPs):

- *TOPparse*, for identification and extraction of header fields and other keywords across all 7 layers of packet headers
- *TOPsearch*, for table lookup and searching operations, typically encountered in classification, routing, policy enforcement, and similar functions
- *TOPresolve*, for packet forwarding based on the lookup results, as well as updating tables, statistics, and other state for functions such as accounting, billing, etc.
- *TOPmodify*, for packet modification

While the architectures of these PEs all revolve around EZchip's superscalar processor architecture, each kind has special features that make it more appropriate for the particular task at hand.

Significant architectures along these lines are the fast pattern processor (FPP) and routing switch processor (RSP), initially of Agere Systems and currently marketed by LSI Logic. Originally, these were separate chips, that

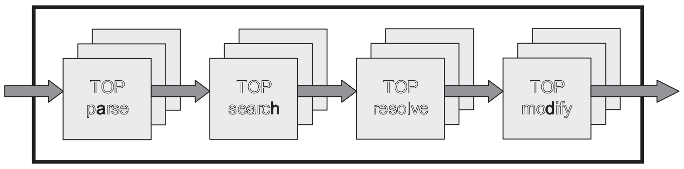

FIGURE 12.12: The concept of the EZchip architecture.

together with the Agere system inteface (ASI) formed a complete chipset for routers and similar systems at the OC-48c design point. Later they were integrated into more compact products, such as the APP550 single-chip solution (depicted in Figure 12.13) for the OC-48 domain and the APP750 two-chip set for the OC-192 domain. The complete architecture is based on a variety of specialized programmable PEs and fixed-function units. The PEs come in several variations:

- The packet processing engine (PPE), responsible for pattern matching operations such as classification and routing. This was the processing core of the original FPP processor.
- The traffic management compute engine, responsible for packet discard algorithms such as RED, WRED, etc.
- The traffic shaper compute engine, for CoS/QoS algorithms.
- The stream editor compute engine, for packet modification.

At the other end of the spectrum we have architectures such as Intel's IXP and IBM's PowerNP, that rely on multiple identical processing engines, that are interchangeable with each other. The PowerNP architecture [3][30] is based on the *dyadic packet processing unit* (DPPU), each of which contains two *picoprocessors*, or *core language processors* (CLPs), supported by a number of custom functional units for common functions such as table lookup. Each CLP is basically a 32-bit RISC processor. For example, the NP4GS3 processor, an instance of the PowerNP architecture, consists of 8 DPPUs (16 picoprocessors total) each of which may be assigned any of the processing steps of the application at hand. The same holds for the IXP and iFlow architectures, that, as mentioned earlier, consist of arrays of identical processing elements. The feature that differentiates this class of architectures from the previous is that for every task that needs to be performed on a packet, the "next available" PE is chosen, without constraints. This is not the case for the EZchip and Agere architectures, where processing tasks are tied to specific PEs.

Finally, we may distinguish a class of architectures that fall in the middle ground, and that includes the C-port and PRO3 processors, among others.

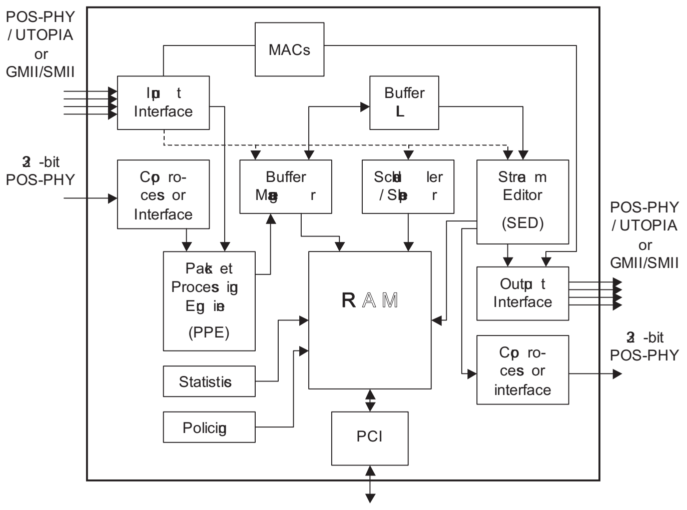

FIGURE 12.13: Block diagram of the Agere (LSI) APP550.

The basis of these architectures is an array of *identical* processing units, each of which consists of a number of *heterogeneous* PEs. Recall the combination of reduced MIPS RISC with the two custom VLIW processors that form the Channel Processor (CP) of the C-port architecture, or the Field Extractor, Packet Processing Engine, and Field Modifier, that together form the reprogrammable pipeline module (RPM) of PRO3. A CP or RPM can be repeated as many times as silicon area allows, for a near-linear increase in performance.

With all heterogeneous architectures, the issue of *load balancing* arises. What is the correct mix of the different kinds of processing elements, and/or, what is the required performance of each kind? Indeed, there is no simple answer that will satisfy all application needs. NPU architects have to resort to extensive profiling of their target applications, based on realistic traffic traces, to determine a design point that will be optimal for a narrow class of applications, provided of course that their assumptions on traffic parameters and processing requirements hold. The broader the target market is for a specific processor, the more difficult it is to attain a single mix of PEs that will satisfy all applications. On the contrary, with homogeneous architectures PEs can be assigned freely to different tasks according to application needs. This may even be performed dynamically, following changing traffic patterns and the mix of traffic flows with different requirements. Of course, for such flexibility one has to sacrifice a certain amount of performance that could be achieved by specialization.

In terms of communication between the processing elements, most NPU architectures avoid fancy and costly on-chip interconnection networks. To justify such a choice, one must consider how packets are processed within an

NPU. Processing of packets that belong to different flows is usually independent. On the other hand, the processing stages for a single packet most often form a pipeline, where the latency between stages is not that critical. Therefore, for most packet processing needs, some kind of shared memory will be sufficient. Note however that usage of an external memory for this purpose would cause a severe bottleneck at the chip I/Os, so on-chip RAM is the norm. For example, in the FPP architecture, a *block buffer* is used to hold 64-byte packet segments (or blocks) until they are processed by the Pattern Processing Engine, the Queue Engine, and other units. In the more recent APP550 incarnation of the architecture, all blocks share access to 3 MB of embedded on-chip DRAM. Similarly, in the PowerNP architecture, packets are stored in global on- and off- chip data stores and from there packet headers are forwarded to the next available PE for processing. No direct communication between PEs is necessary in the usual flow of processing.

There are of course more elaborate communication schemes than the above, with most noteworthy probably the IXP case. In this architecture, PEs are divided in two 8-PE clusters. The PEs of each cluster communicate with each other and with other system components over two buses (labeled D and S). No direct communication between the two clusters is possible. Each PE (Figure 12.14) has a number of registers, called *transfer registers*, dedicated to inter-PE communication. By writing to an output transfer register, a PE can directly modify the corresponding input transfer register of another PE. Furthermore, another set of registers is dedicated to nearest neighbor communication. With this scheme, each PE has direct access to the appropriate register of its neighbor. In this way, a very efficient ring is formed.

In the C-port family, a hierarchy of buses is also used. Three different buses, with bandwidths ranging from 4.2 to 34.1 Gbits/sec (on the C-5e), are used to interconnect all channel processors and other units with each other.

12.3.2 Multi-Threading Support

Turning now to the microarchitecture of the individual PEs, a prevailing trend in NPU architectures is multi-threading. The reason that most NPU vendors have converged to this technique is that it offers a good method to overcome the unavoidably long latency of certain operations. Table lookup is a characteristic one. It is often handled by specialized coprocessors and can take a large number of clock cycles to complete. But as with all complex SoCs, even plain accesses to external memories, such as the packet buffer, incur a significant latency. Multi-threading allows a processing element to switch to a new thread of execution, typically processing a different packet, every time a long-latency operation starts. It is important to note here that the nature of most network processing applications allows multi-threading to be very effective, since there will almost always be some packet waiting to be processed, and each packet can be associated with a thread. So, ready-to-run threads will almost always be available and most of the time long latency operations of one or more threads will overlap with processing of another thread. In this way, processing cycles will almost never get wasted waiting for long-running operations to complete.

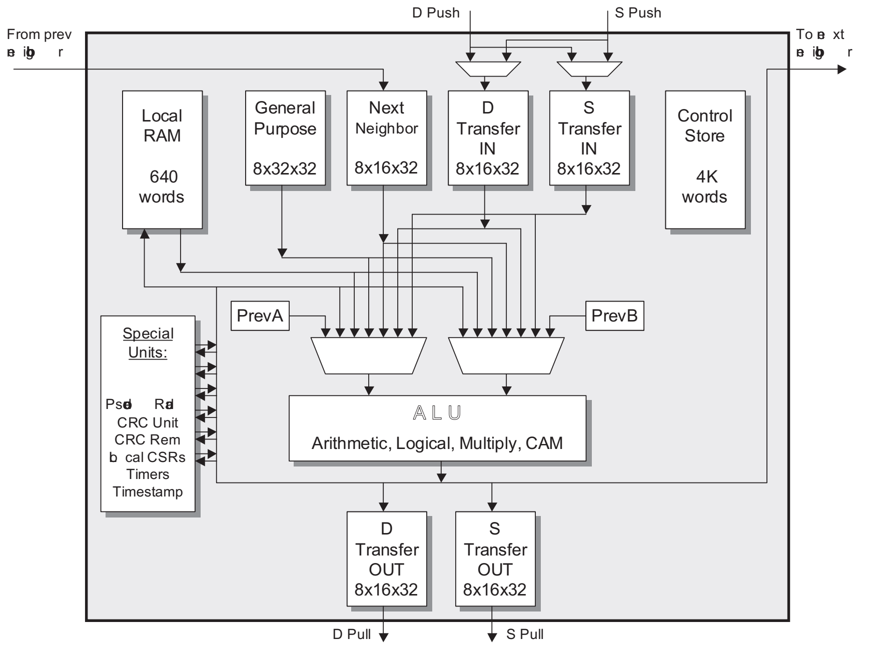

FIGURE 12.14: The PE (microengine) of the Intel IXP2800.

For multi-threading to be effective, switching between threads must be possible with very little or no overhead. Indeed, many network processor vendors claim zero-overhead thread switching. To make this possible, the structure of the PE is augmented with multiple copies of all execution state. By the term *state* we define the content of registers and memory, as well as the program counter, flags and other state bits, depending on the particular architecture. So, multi-threaded PEs typically have register files partitioned into multiple banks, one per supported thread, while local memory may also be partitioned per thread. Events that trigger thread switching can be a request to a coprocessor or an external memory access. On such an event, the current thread becomes inactive, a new thread is selected among those ready for execution, and the appropriate partition of the register file and related state is activated. When the long-running operation completes, the stalled thread will become ready again and get queued for execution.

A critical design choice is the number of supported threads per PE. If the PE does not directly support enough threads in hardware, the situation will often arise that all supported threads are waiting for an external access, in which case the processing cycles remain unused. Processors with shorter cycle times and more complex coprocessors (requiring longer to complete) or a slower external memory system will require more threads. On the other hand, the cost of supporting many threads can have a significant impact on both die area and cycle time. Therefore, this parameter must be chosen very judiciously, based on profiling of target applications and performance simulations of the planned architecture.

Most industrial designs offer good examples of multi-threading: Each picoprocessor in IBM's NP4GS3 supported two threads, a number that was apparently found insufficient and later raised to four in the more recent 5NP4G (marketed by HiFn). Threads also share a 4 KB local memory available within each DPPU of the NP4GS3, each one having exclusive access to a 1 KB segment. The iPP and IXP architectures are very similar with respect to multi-threading; each architecture supports eight threads per PE, each with its register file partition and other state. Thread switching is performed with zero overhead, when long-running instructions are encountered along the thread's execution path. Such instructions may be external memory accesses or complex functions executed on a coprocessor. The programmer also has the possibility to relinquish control by executing special instructions that will cause the current thread to sleep, waiting for a specific event. Finally, noteworthy is the case of the FPP, whose single PE supports up to 64 threads!

The PRO3 processor follows a different approach for overlapping processing with slow memory accesses. The FEX-PPE-FMO pipeline is organized in such a way that these processing engines almost always work out of local memory. The PPE's register file has two banks. One of them can be accessed directly by either FEX or FMO, at the same time that the PPE is executing, using the other bank. In addition, the PPE's local memory has two ports, one of which can be accessed by an external controller. When a packet arrives at the RPM, the FEX extracts all necessary fields from its headers, under program control. It then writes the values of these fields into one bank of the PPE register file. To retrieve per-flow state from off-chip memory, a flow identifier (FlowId) is constructed from the packet header, that is used as index to memory. State retrieved thus is written into the PPE's local memory over its external port. These actions can take place while the PPE is still processing the previous packet. When it finishes, the PPE does not need to output the results explicitly, since the FMO can pull the results directly out of the PPE's register file. A data I/O controller external to the PPE will also extract data from the PPE's local memory to update flow state in the off-chip RAM. All that the PPE needs to do is to switch the two partitions of the register file and local RAM and restart executing. The relevant header fields and flow state will already be present in its newly activated partitions of the register file and local RAM respectively. In this way, data I/O instructions are eliminated from the PPE code and computation largely overlaps with I/O (output of the previous packet's results and input of the next packet's data). With the PPE working on local memory (almost) all the time, there is very little motivation for multi-threading support. So, PRO3 PPEs do not need to support more than one thread.

12.3.3 Specialized Instruction Set Architectures

Finally, the instruction set architecture (ISA) is another area where vendors tend to innovate and differentiate from each other. While some vendors rely on more-or-less standard RISC instruction sets, it is recognized by many that this is not an efficient approach; instead, an instruction set designed from scratch and optimized for the special mix of operations common in packet processing can give a significant performance edge over a simple RISC ISA. This is easy to comprehend if one considers that RISC instruction sets have resulted from years of profiling and analyzing *general-purpose* computing applications; it is only natural to expect that a similar analysis on *networking* applications should be the right way to define an instruction set for an NPU.

Based on the above rationale, many NPU vendors claim great breakthroughs in performance, solely due to such an optimized instruction set. AMCC has dubbed its ISA NISC (network instruction set computing) in analogy to RISC. EZchip promotes its Task Optimized Processing Core technology, with customized instruction set and datapath for each packet processing stage. Interestingly, both vendors claim a speedup over RISC-based architectures in the order of 10 times. Finally, Silicon Access, with its iFlow architecture, also based on a custom instruction set, claimed double the performance of its nearest competitor.

One can distinguish two categories of special instructions encountered in NPU ISAs: those that have to do with the coordination of multiple PEs and multiple processing threads working in parallel, and those that perform packet processing-oriented data manipulations. In the first category one can find instructions for functions such as thread synchronization, mutual exclusion, inter-process (or -thread) communication, etc. We can mention for example support in the IXP ISA for atomic read-modify-write (useful for mutual exclusion) and events, used for signalling between threads. Instructions that fall in this first category are also encountered in parallel architectures outside of the network processing domain. In the following we will focus on the data manipulation operations.

Arguably the most common kinds of operations in packet processing have to do with header parsing and modification: extraction of bit fields of arbitrary length from arbitrary positions in the header for the parsing stage, on packet ingress, or similar insertions for the modification stage, on packet egress. Many NPU architectures cater to accelerate such operations with custom instructions. For example, the IXP combines shifting with logical operations in one cycle, to speed-up the multiple shift-and-mask operations needed to parse a header. Also, the iFlow architecture supports single-cycle insertion and extraction of arbitrary bit fields. The same is true for the Field Extractor and Field Modifier in the PRO3 architecture.

Multi-way branches are also common when parsing fields such as packet type, or encoded protocol identifiers. With standard RISC instruction sets, a wide switch statement is translated into many sequential compare-and-branch statements. Custom ISAs accelerate this kind of code by special support for conditional branches. Silicon Access claimed to be able to speed up such cases by up to 100 times, with a technology dubbed *massively parallel branch acceleration* that allows such a wide switch to be executed in only two clock cycles. As another example, the IXP microengine includes a small CAM that can be used to accelerate multi-way branches, by allowing up to 16 comparisons to be performed in parallel, providing at the same time a branch target.

Predicated execution is another branch optimization technique, that is actually borrowed from the DSP world. It allows execution of certain instructions to be enabled or disabled based on the value of a flag. In this way, many conditional branch operations are avoided, something that can speed up significantly tight loops with many short if-then-else constructs. The CLP processor of the PowerNP architecture is an example of such an instruction set.

Finally, many architectures provide instructions for tasks such as CRC calculation and checksumming (1's complement addition), evaluation of hash functions, pseudorandom number generation, etc. Another noteworthy addition is support in the IXP architecture for efficient linked list and circular buffer operations (insert, delete, etc). Given that the use of such structures in networking applications is very common, such hardware support has a significant potential for overall code speedup.

12.4 Address Lookup and Packet Classification Engines

The problem of packet classification is usually the first that has to be tackled when packets enter a router, firewall, or other piece of network equipment. Before classification the system has no information regarding how to handle incoming packets. To maintain wire speed operation, it has to decide very quickly what to do with each new packet received: queue it for processing, and if so, to which queue? Discard it? Any other possibility? The classifier is the functional unit that will inspect the packet and provide the necessary information for such decisions.

In general, a classifier receives an unstructured stream of packets and by applying a configurable set of rules it splits this stream into parallel flows of packets, with all packets that belong to the same flow having something in common. The definition of this common feature is arbitrary. Historically it has been the destination port number (where classification served solely the purpose of forwarding). But more recently it may represent other notions, such as same QoS requirements, or type of security processing, or other. Whatever this common characteristic is, it implies that all packets of a flow will be processed by the router in the same manner, at least for the next stage (or stages) of processing. The decision as to how to classify each incoming packet depends on one (rarely) or multiple (more commonly) fields of the packet header(s) at various layers of the protocol hierarchy.

Classification is not an easy problem, especially given that it has to be performed at wire speed. Even in the case of simple route lookup based on the packet's destination IP address (probably the simplest special case of the problem) it is not trivial. Consider that an IPv4 address is 32 bits wide, with normally up to 24 bits used for routing. A naïve table implementation would contain 2^{24} entries, something prohibitive. However, such a table would be quite sparse, motivating implementations based on various kinds of data structures. The size of such a table would be a function of the active (valid) entries only. Unfortunately, this is still a large number. Up-to-date statistics maintained by [1] show that as of this writing, the number of entries in the Internet's core routers (known as the BGP table, from the Border Gateway Protocol) has exceeded 280,000 and is still rising. Searching such a table at wire speed at 10 Gbps is certainly a challenge; assuming a flow of minimum-size IP packets, only 32 nsec are available per search. Consider now that this is only a one-dimensional lookup. In more demanding situations classification has to be based on multiple header fields. Typical is the quintuple of source and destination IP addresses, source and destination port numbers, and layer 4 protocol identifier, often used to define a flow. Finally, such tables have to be updated dynamically in large metropolitan and wide area networks, more than 1000 times per second.

Classification also appears further down the processing pipeline, depending on the application. Classification based on the aforementioned quintuple is applicable to tasks such as traffic management, QoS assurance, accounting, billing, security processing, and firewalls, just to name a few. Classification can even be performed on packet payload, for example on URLs appearing in an HTTP message, for applications such as URL filtering and URL-based switching.

Formally, the problem of classification can be stated as follows: For any given packet, a search key or lookup key is defined as an arbitrary selection of N header fields (an *N-tuple*). A rule is a tuple of values, possibly containing wildcards, against which the key has to be matched. A *rule database* is a prioritized list of such rules. The task of classification is to find the highest priority rule that matches the search key. In most cases, the index of the matching rule is used as the *flow identifier* (flowID) associated with all packets that match the same rule. So, each rule defines a *flow*.

Wildcards usually take one of two forms: (i) prefixes, usually applicable to IP addresses. For example, the set of addresses 192.168.*.* is a 16-bit prefix. This is an effect of the way Classless Interdomain Routing (CIDR) [11] works and gives rise to a variety of longest-prefix matching (LPM) algorithms and (ii) ranges, most commonly used with port numbers, such as 100-150.

12.4.1 Classification Techniques

The simplest and fastest way to search a rule database is by use of a content-addressable memory (CAM). Indeed, CAMs are used often in commercial classification engines, even though they have certain disadvantages. In contrast to a normal memory, that receives an address and provides the data stored in that address, a CAM receives a data value and returns the address where this value is found. The entire array is searched in parallel, usually in a single clock cycle, the matching locations are identified by their address, and a priority encoder resolves potential multiple matches. One or more match addresses may be returned.

The growing importance of LPM matching has given rise to *Ternary CAMs*, or TCAMs, that support wildcarding. For every bit position in a TCAM, two actual bits are used: a care/don't care bit and the data bit. All the care/don't care bits of a memory address form a mask. In this way prefixes can be easily specified. For example the IP address prefix 192.168.*.* can be specified with data value 0xC0A80000 and mask 0xFFFF0000.

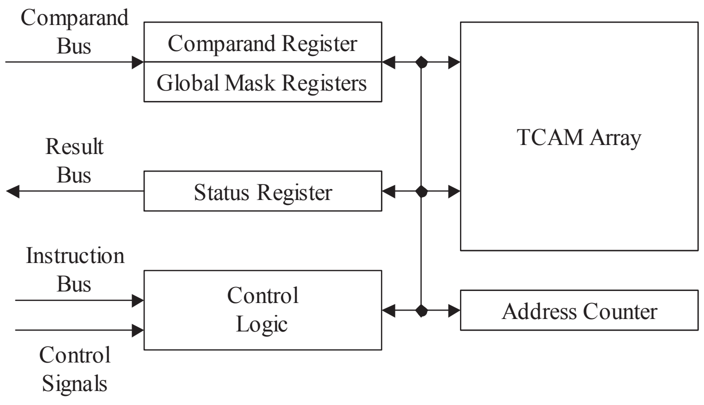

FIGURE 12.15: TCAM organization [Source: Netlogic].

Searching with a CAM becomes trivial. One needs only concatenate the relevant header fields, provide those to the CAM, and wait for the match address to be returned. The main disadvantage of CAMs (and even more so of TCAMs) is the silicon area required, which is several times larger than that of simple memory. This gives rise to high cost, limited overall capacity, and impact on overall system dimensions. Furthermore, the parallel search of the memory array causes a high power dissipation. In spite of these problems, TCAMs are not uncommon in commercial systems. They are certainly more appropriate in highest throughput systems (such as OC-48 and OC-192 core routers), which are also the least cost-sensitive.

For the cases where a TCAM is not deemed cost-efficient, a variety of algorithmic approaches have been proposed and applied in many practical systems.

Most of these approaches store the rule database in SRAM or DRAM in some kind of pointer-based data structure. A search engine then traverses this data structure to find the best-matching rule. In practical systems, this search engine may be fixed logic, although programmable units are also common, for reasons of flexibility.

In the following we briefly review two representative techniques. A good survey of algorithms can be found in [15]. When examining such algorithms, one needs to keep in mind that in addition to lookup speed, such algorithms must be evaluated for the speed and ease of incremental updates, and memory size and cost (e.g., whether they require SRAM or DRAM).

12.4.1.1 Trie-based Algorithms

Many of the most common implementations of the classifier database are based on the *trie* data structure [23]. A trie is a special kind of tree, used for creating dictionaries for languages with arbitrary alphabets, that is quite effective when words can be prefixes of other words (as is the case of IP address prefixes). When the alphabet is the set of binary digits, a trie can be used to represent a set of IP addresses and address prefixes. Searching for a prefix in a single dimension, as in the case of route lookup, is simple: just traverse the tree based on the digits of the search key, until either a match is found or the key characters are exhausted. Obviously, nodes lower in the tree take precedence, since they correspond to longer matches. The problem gets more interesting when multidimensional searches are required.

A *hierarchical* or *multilevel* trie can be thought of as a three-dimensional trie, where the third dimension corresponds to the different fields of an N-dimensional key. Lookup involves traversing all dimensions in sequence, so the lookup performance of the basic hierarchical trie search is O(Wd), where W is the key width and d the number of dimensions. The storage requirements are O(NdW), with N the number of rules. Finally, incremental updates are possible with complexity $O(d^2W)$. Details on the construction and lookup of hierarchical tries can be found in references such as [15].

Many variations of the basic algorithm have also been proposed. For example, for two-dimensional classifiers, the *grid-of-tries* algorithm [41] enhances the data structure with some additional pointers between nodes in the second dimension tries, so that no backtracking is needed and the search time is reduced to O(W). However, this comes at the expense of difficult incremental updates, so rebuilding the database from scratch is recommended. So, this algorithm is appropriate for relatively static classifiers only.

12.4.1.2 Hierarchical Intelligent Cuttings (HiCuts)

This is representative of a class of algorithms based on the geometric interpretation of classifiers. A two-dimensional classifier can be visualized as a set of rectangles contained in a box that is defined by the overall ranges of the two dimensions. For example, Figure 12.16 defines a classifier:

Rule	X	Y
R1	0*	*
R2	1*	*
R3	0*	1*
R4	10*	00*
R5	*	010
R6	000	00*
R7	001	00*

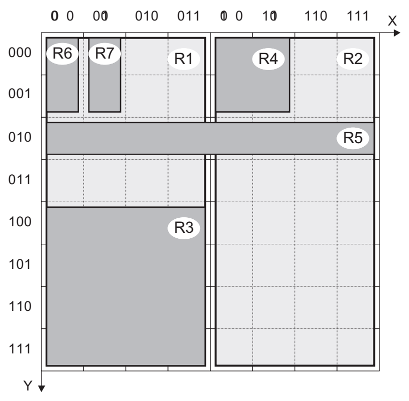

FIGURE 12.16: Mapping of rules to a two-dimensional classifier.

While we use here a two-dimensional example for the purpose of illustration, the algorithm generalizes to any number of dimensions. HiCuts [14] constructs a decision tree based on heuristics that aim to exploit the structure of the rules. Each node of the tree represents a subset of the space. A cut, determined by appropriate heuristics, is associated with each node. A cut partitions the space along one dimension into N equal parts, creating N children, each of which represents one Nth of the original box. Each node is also associated with all rules that overlap fully or partially with the box it represents. Cutting proceeds until all leaf nodes contain at most B rules, where B is a tunable parameter trading storage space for lookup performance. To match a given search key, the algorithm traverses the decision tree guided by the bits of the key, until it hits a leaf node. Then, the B or fewer rules that leaf contains are searched sequentially to determine the best match.

12.4.2 Case Studies

Finally we review some of the most representative classification/table lookup engines in the industry.

PowerNP. The Dyadic Packet Processing Unit (DPPU) of the PowerNP architecture [3] contains two RISC cores, along with two *Tree Search Engines* (TSEs), together with other coprocessors. The TSE is a programmable unit that supports table lookup in three modes: full match (for looking up structures like MAC address tables), longest-prefix match (for example for layer 3 forwarding) and software-managed trees, the most general kind of search. This last mode supports all the advanced search features, such as general N-tuple matching, and support for arbitrary ranges in any dimension (not just prefixes).

Operation of the TSE starts with a RISC core constructing the search key from the appropriate header fields. Then, it issues a request to one of the two TSEs of the DPPU to execute the search. The TSE first consults the *LuDefTable* (Lookup Definition Table), an on-chip memory that contains information about the available tables (where they are stored, the kind of search to do, key sizes, tree formats, etc). The TSE also has access to the system's *control store* (control memory) where tables are stored, among other data. The control store is a combination of on-chip memory with off-chip DDR SDRAM and ZBT SRAM (in the newer NP4GX, Fast Cycle RAM (FCRAM) is also used).

Typical performance numbers for the TSE of the NP4GS3 are from 8 to 12 million searches per second, depending on the type of search, a rate sufficient to support basic processing at an OC-48 rate (2.5 Gbps), with minimum size IP packets and one lookup per packet. In case higher performance is needed, the NP4GS3 also supports external CAM.

Agere. The primary role of Agere's (currently LSI Logic's) Fast Pattern Processor [30] is packet header parsing and classification. The Packet Processing Engine (PPE), the main programmable unit of the FPP, is programmed in Agere's own *Functional Programming Language* (FPL). As its name implies, FPL is a functional language, which is very appropriate for specifying patterns to be matched. Supposedly, it also generates very compact machine code, at least for the kinds of tasks encountered in packet classification. The FPP also uses a proprietary, patented search technique, that Agere has dubbed Pattern Matching Optimization. This technique places emphasis on fast lookups, which are executed in time bounded by the length of the key (pattern) and not by the size of the database.

The FPP processes data in 64-byte blocks. Complete processing of a packet involves two steps, or passes. When packets enter the FPP, they are first segmented to blocks and stored in the external packet buffer. At the same time, the first block of each packet is loaded into a context, an on-chip storage area that maintains short-term state for a running thread. With 64 threads supported in hardware, there are 64 contexts to choose from. Once basic first-pass processing is done, the packet is assigned to a *replay queue*, getting in line for the second pass. When a context is available it is loaded and the second pass starts. Once the second pass is over, the packet is sent downstream to the RSP, followed by the classification results that the PPE retrieved.

While the original FPP relied on SRAM for classification database storage, newer incarnations of the architecture, such as the 10 Gbps APP750NP, replace this with FCRAM, reducing the cost and at the same time achieving better performance than would be possible with regular DRAM.

Silicon Access. Silicon Access introduced the iFlow product family [30], consisting of several chips: packet processor, traffic manager, accountant (for billing etc) and not one but two search engines: the Address Processor (iAP) and the Classifier (iCL). Even though the company did not survive the slowdown of core network rollouts in the early 2000s, the architecture has several interesting features that makes it worth examining.

The two search engines are designed for different requirements. The Address Processor can perform pipelined, full or longest prefix matching operations on on-chip tree-based tables with keys ranging from 48 to 144 bits wide. On the other hand, the Classifier is TCAM-based and performs general range matching with keys up to 432 bits long. So, the iAP is more appropriate for operations like address lookup, while the more general classification problem is the task of the iCL.

The innovation of Silicon Access in the design of the iFlow chipset is undoubtedly the use of wide embedded DRAM, an architectural choice that in many applications eliminates the need for external CAMs and SRAMs, and even reduces the pressure on external DRAM. The two search engines rely entirely on on-chip memory. The iAP has a total of 52 Mbits of memory, holding 256K prefixes up to 48 bits long, 96 bits of associated data memory per entry, and a smaller 8K by 256 per-next-hop associated data memory. The iCL's 9.5 Mbits of total memory are organized as 36K entries by 144 bits of TCAM plus 128 bits associated data per entry. Of course, in both systems multiple table entries can be combined to cover each device's maximum key width. The much smaller amount of total memory in the iCL is unavoidable, given that most of it is organized as a TCAM, with much lower density than plain RAM. It is also noteworthy that all embedded memory in these devices is ECC protected, which makes them effective for high reliability applications. In terms of performance, the devices are rated at 100 Msps (iCL) and 65 Msps (iAP), allowing up to three or two, respectively, searches per minimum-size IP packet on a 10 Gbps link. The embedded memory-based architecture of iAP and iCL is of course both a curse and a blessing: on one hand it reduces the chip count, cost, and power dissipation of the system; on the other, it places a hard limit on the table sizes that can be implemented. Fortunately, multiple devices can be combined to form larger tables, although this is unlikely to be cost-effective.

In the following we give a few details about the organization and operation of the iAP, the more interesting of the two devices from an algorithmic standpoint [34]. The device has a ZBT SRAM interface, over which it is attached to a network processor, such as the iPP. The network processor performs regular memory writes to issue requests and memory reads to retrieve the results. iAP's operation is pipelined and with fixed latency, independent of the number of entries, prefix length or key length: it can start a new lookup every 2 clock cycles (at 133MHz), with a latency of 26 cycles.

The search algorithm, which is hardwired in fixed logic, takes advantage of the very wide on-chip RAMs that allow many entries to be read out in parallel and an equal number of comparisons to be made simultaneously. Prefixes are stored in RAM in ascending order, with shorter prefixes treated as larger than longer ones; for example, 11011* is larger than 1101101*. A three-level B-tree

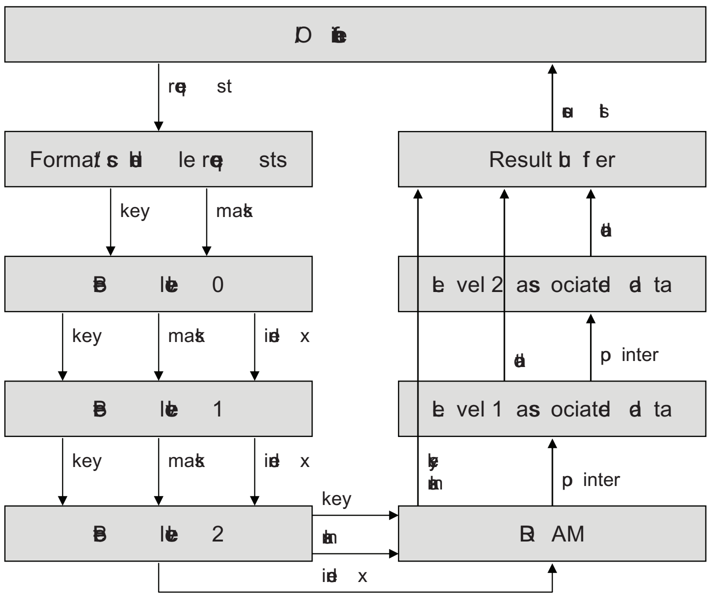

FIGURE 12.17: iAP organization.

in on-chip SRAM provides pointers to the complete list of prefixes, stored in on-chip DRAM. Traversing the three levels of the B-tree results in a pointer to a small subset of the prefix list. A small number of parallel comparisons there allows the correct prefix to be located.

We should also note that the architecture of the iAP allows table maintenance (insertions and deletions of prefixes) to be performed in parallel with the searches. The iAP can support up to 1 million updates per second, consuming only about 20 percent of the search bandwidth.

EZchip. EZchip stresses the implementation of classifier tables in DRAM, which helps reduce system cost, chip count and power dissipation. A second feature emphasized is the support for long lookups (for arbitrary length strings, such as URLs).

In EZchip's heterogeneous architecture, the processing engine (Figure 12.18) dubbed TOPsearch is the one responsible for table lookup operations [10]. This is primarily a fixed-logic engine, with a minimal instruction set designed to support chained lookups, where the result of one lookup, possibly combined with additional header fields, is used as the key for a new lookup. TOPsearch supports three types of lookups: direct, hash, and tree-based. In the latter case, the optimization employed to make operation at high link rates possible is to store the internal nodes of the tree on-chip in embedded DRAM, and the leaf nodes in external DRAM. The embedded memory organization, with a 256-bit wide interface, allows up to three levels of the tree to be traversed with a single memory access. Put together with the shorter access time of on-chip DRAM, this architecture provides a significant speedup compared with the more common external memory organization.

Finally, parallelism is employed: EZchip NPUs include a number of TOPsearch engines, that can work concurrently on different packets. With all the above optimizations, tables with over one million entries can be searched at up to a 30 Gbps link rate with the NP-3 NPU.

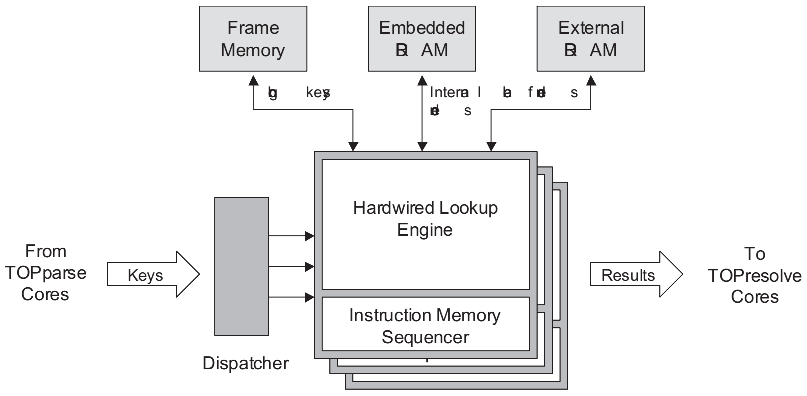

FIGURE 12.18: EZchip table lookup architecture.

Third-party search engines. A limited number of vendors specialize in search engines, without a full network processing chipset in their portfolio. The standardization of a coprocessor interface for search engines by the Network Processing Forum[1], dubbed LA-1(b) [13], helps in the integration of such third-party devices into systems built around NPU families of most major vendors.

Typical is the case of Netlogic Microsystems, which has been among the leading suppliers of CAM and TCAM devices. The obvious path toward on-chip integration has been to incorporate table lookup and maintenance logic into the TCAM device, thus transforming what was only table storage into a self-contained search engine. A number of variations on the theme are provided, ranging from plain address lookup engines for IP forwarding, to layer four classification engines supporting N-tuple lookup, all the way to layer seven processors for "deep packet inspection", for applications such as URL matching and filtering, virus signature recognition, stateful packet inspection, etc.

With this kind of specialized search engine, key matching is becoming increasingly sophisticated. For example, the above mentioned engines sup-

[1]Later merged into the Optical Internetworking Forum

port regular expression matching, an additional step up in complexity and sophistication from the longest-prefix matching and range lookup that we have discussed so far. In fact, a long search key may span the payload of more than one packet. The capability of on-the-fly inspection of packets all the way up to layer seven, combined with such sophisticated matching, leads to new applications, such as intrusion detection, general malware detection, application-based switching, etc. Standardization of interfaces, such as the LA-1, certainly fosters innovation in this field, since it allows more players to enter the market with alternative architectures.

12.5 Packet Buffering and Queue Management Engines

Most modern networking technologies (like IP, ATM, MPLS etc.) share the notion of connections or flows (we adopt the term *flow* hereafter) that represent data transactions in specific time spans and between specific end-points in the network for the implementation of networking applications. Furthermore scheduling among multiple per port, QoS and CoS queues requires the discrimination of packet data and the handling of multiple data flows with differentiated service requirements. Depending on the applications and algorithms used, the network processor typically has to manage thousands of flows, implemented as packet queues in the processor packet buffer [27]. Therefore, effective queue management is key to high-performance network processing as well as to reducing development complexity. In this section we focus on the review of potential implementations within a NPU architecture and performance evaluation of queue management, which is performed extensively in network processing applications and show how HW cores can be used to offload completely this task from other processing elements.

The requirements with regard to memory management implementations in networking applications stem from the fact that data packets need to be stored in an appropriate queue structure either before or after processing and be selectively forwarded. These queues of packets need to at least serve the first-in-first-out (FIFO) service discipline, while in many applications flexible access to their data is required (in order to modify, move, delete packets or part of a packet, which resides in a specific position in the queue, e.g., head or tail of the queue etc.). In order to efficiently cope with these requirements several solutions based on dedicated hardware have been proposed initially targeting high-speed ATM switching where the fixed ATM cell size favored very efficient queue management [39][33] [46] and later extended to management of queues of variable-size packets [18]. The basic advantage of these implementations in hardware is of course the higher throughput with modest implementation cost. On the other hand the functions they can provide (e.g., single versus double linked lists, operations in the head/tail of the queue, copy operations etc.)

needs to be selected carefully at the beginning of the design. Several trade-offs between dedicated hardware and implementations in software have been exposed in [48], in which specific implementations of such queue management schemes in ATM switching applications are examined.

As in many other communication subsystems, memory access bandwidth to the external DRAM-based packet data repository is the scarcest resource in NPUs. For this reason, the NPU architecture must be designed very carefully to avoid unnecessary data transfer across this memory interface. In an NPU architecture, each packet byte may traverse the memory interface up to four times, e.g., when encryption/decryption or deep packet parsing functions are performed. This is also the case for short packets such as TCP/IP acknowledgments, where the packet header is the entire packet, in order to perform the following operations: (a) write packet to buffer on ingress, (b) read header/packet into processing engines, (c) write back to memory, and (d) read for egress transmission.

This means that for small packets, which typically represent 40 percent of all Internet packets, the required memory interface capacities amount to 10, 40, or 120 Gb/s for OC-48, OC-192, or OC-768, respectively. Even the lowest of these values, 10 Gb/s, exceeds the access rate of todays commercial DRAMs. Complex memory-interleaving techniques that pipeline memory access and distribute individual packets over multiple parallel DRAM (dynamic RAM) chips can be applied for 10 Gb/s and possibly 40 Gb/s memory subsystems. At 120 Gb/s, todays 166 MHz DDR (double-data-rate) SDRAMs would require well over 360-bit-wide memory interfaces, or typically some 25 DDR SDRAM chips.

Several commercial NPUs follow a hybrid approach targeting the acceleration of memory management implementations by utilizing specialized hardware units that assist specific memory access operations, without providing a complete queue management implementation. The first generation of the Intel NPU family, the IXP1200, initially provided an enhanced SDRAM unit, which supported single byte, word, and long-word write capabilities using a read-modify-write technique and may reorder SDRAM accesses for best performance (the benefits of this will also be explored in the following section). The SRAM Unit of the IXP1200 also includes an 8-entry push/pop register list for fast queue operations. Although these hardware enhancements improve the performance of typical queue management implementations they cannot keep in pace with the requirements of high-speed networks. Therefore the next generation IXP-2400 provides high-performance queue management hardware that automates adding data to and removing data from queues [40]. Following the same approach the PowerNP NP4GS3 incorporates dedicated hardware acceleration for cell enqueue/dequeue operations in order to manage packet queues [3]. The C-Port/Motorola C-5 NPU also provided memory management acceleration hardware [6], still not adequate though to cope with demanding applications that require frequent access to packet queues. The next-generation Q-5 Traffic Management Coprocessor provided dedicated hardware designed to support traffic management for up to 128K queues at a rate of 2.5 Gbps [18]. In the rest of this section we review the most important performance requirements evaluating a set of alternative implementations that dictate the basic design choices when assigning specific tasks to embedded engines in a multi-core NPU implementation.

12.5.1 Performance Issues

12.5.1.1 External DRAM Memory Bottlenecks

A crucial design decision at such high rates is the choice of the buffer memory technology. Static random access memory (SRAM) provides high throughput but limited capacity, while DRAM offers comparable throughput and significantly higher capacity per unit cost; thus, DRAM is the prevalent choice among all NPUs for implementing large packet buffering structures. Furthermore, among DRAM technologies, DDR SDRAM is becoming very popular because of its high performance and affordable price. DDR technology can provide 12.8 Gbps peak throughput by using a 64-bit data bus at 100 MHz with double clocking (i.e., 200 Mbps/pin). A DIMM module provides up to 2 GB total capacity and it is organized into four or eight banks to provide interleaving (i.e., to allow multiple parallel accesses). However, due to bank-pre-charging periods (during which the bank is characterized as busy) successive accesses must respect specific timing requirements. Thus, a new read/write access to 64-byte data blocks to the same bank can be inserted every four clock-cycles, i.e., every 160 ns (with an access cycle of 40 ns). When a memory transaction tries to access a currently busy bank, we say that a bank conflict has occurred. This conflict causes the new transaction to be delayed until the bank becomes available, thus reducing memory utilization. In addition, interleaved read and write accesses also cause loss to memory utilization because they create different access delays. Thus, while the write access delay can be as low as 40 ns and the read access delay 60 ns, when write accesses occur after read accesses, the write access must be delayed by one access cycle.

It is worth demonstrating the impact of the above implications in DDR-DRAM performance in the overall aggregate throughput that can be provided under usual access patterns following the methodology presented in [36]. The authors in [36] simulated a behavioral model of a DDR-SDRAM memory under a random access pattern and estimated the impacts of bank conflicts and read-write interleaving on memory utilization. The results of this simulation for a range of available memory banks (1 to 16) are presented in the two left columns of Table 12.1.

The access requests assume aggregate accesses from two write and two read ports (a write and a read port from/to the network, a write and a read port from/to the internal processing element (PE) array). By serializing the accesses from the four ports in a simple/round-robin order (i.e., without optimization) the throughput loss presented in Table 12.1 is achieved. However, by

TABLE 12.1: DDR-DRAM Throughput Loss Using 1 to 16 Banks

Banks	No Optimization		Optimization	
	Throughput Loss		Throughput Loss	
	Bank conflicts	Bank conflicts + write-read interleaving	Bank conflicts	Bank conflicts + write-read interleaving
1	0.750	0.75	0.750	0.750
2	0.647	0.66	0.552	0.660
3	0.577	0.598	0.390	0.432
4	0.522	0.5	0.260	0.331
5	0.478	0.48	0.170	0.290
6	0.442	0.46	0.100	0.243
7	0.410	0.42	0.080	0.220
8	0.384	0.39	0.046	0.199
9	0.360	0.376	0.032	0.185
10	0.338	0.367	0.022	0.172
11	0.321	0.353	0.018	0.165
12	0.305	0.347	0.012	0.159
13	0.289	0.335	0.010	0.153
14	0.275	0.33	0.007	0.148
15	0.264	0.32	0.004	0.143
16	0.253	0.317	0.003	0.139

scheduling the accesses of these four ports in a more efficient manner, a lower throughput loss is achieved since a reduction in bank conflicts is possible. A simple way to do this is to effectively reorder the accesses of the four ports to minimize bank conflicts. The information for bank availability in order to appropriately schedule accesses is achieved by keeping the memory access history (i.e., storing the last three accesses). In case that more than one accesses are eligible (belong to a non-busy bank), the scheduler selects one of the eligible accesses in round-robin order. If no pending access is eligible, then the scheduler sends a no-operation to the memory, losing an access cycle. The results of this optimization are presented in the right side of Table 12.1. Assuming organization of eight banks, the optimized scheme reduces throughput loss by 50 percent with respect to the un-optimized scheme. Thus, it is evident that only a percentage of the nominal 12.8 Gbps peak throughput of a 64-bit/100 MHz DDR-DRAM can be utilized and the design of the memory controller must be an integral part of the memory management solution.

12.5.1.2 Evaluation of Queue Management Functions: INTEL IXP1200 Case

As described above, the most straightforward implementation of memory management in NPUs is based on software executed by one or more on-chip microprocessors. Apart from the memory bandwidth that was examined in isolation in the previous section, a significant factor that affects the overall performance of a queue management implementation is the combination of the processing and communication latency (communication with the peripheral memories and memory controllers) of the queue handling engine (either generic processor or fixed/configurable hardware) and the memory response latency. Therefore the overall actual performance can only be evaluated at system level. Using Intel's IXP1200 as an example representing a typical NPU architecture, the authors in [36] have also presented results regarding the maximum throughput that can be achieved when implementing memory management in IXP1200 software.

The IXP1200 consists of six simple RISC processing microengines [28] running at 200 MHz. According to [36], when porting the queue management software to the IXP RISC-engines, special care should be given so as to take advantage of the local cache memory (called scratch memory) as much as possible. This is because any accesses to the external memories use a very large number of clock cycles. One can argue that using the multi-threading capability of the IXP can hide this memory latency. However, as it was proved in [48], the overhead for the context switch, in the case of multi-threading, exceeds the memory latency and thus this IXP feature cannot increase the performance of the memory management system when external memories should be accessed. Even by using a very small number of queues (i.e., fewer than 16), so as to keep every piece of information in the local cache and in the IXPs registers, each microengine cannot service more than 1 million packets per second (Mpps). In other words, the whole of the IXP cannot process more than 6 Mpps. Moreover, if 128 queues are needed, and thus external memory accesses are necessary, each microengine can process at most 400 Kpps. Finally, for 1K queues the peak bandwidth that can be serviced by all six IXP microengines is about 300 Kpps [40]. The above throughput results are summarized in Table 12.2.

TABLE 12.2: Maximum Rate Serviced When Queue Management Runs on IXP 1200

No. of Queues	1 Microengine	6 Microengines
16	956 Kpps	5.6 Mpps
128	390 Kpps	2.3 Mpps
1024	60 Kpps	0.3 Mpps

12.5.2 Design of Specialized Core for Implementation of Queue Management in Hardware

Due to the performance limitations identified above, the only choice to achieve very high capacity (mainly in NPUs targeting core network systems and high-speed networking applications) is to implement dedicated embedded cores to offload the other PEs from queue management tasks. Such cores are implemented either as fixed hardware engines, designed specifically to accelerate the task of packet buffering and per-flow queuing, or as programmable HW cores with limited programmability extending to a range of operations indexed by means of an OPCODE that can be executed. In the remainder of this section we present the micro-architecture and performance details of such a specifically designed engine (originally presented in [36]) designed as a task-specific embedded core for NPUs supporting most of the requirements for queue and buffer management applications. The maintenance of queues of packets per flow in the design presented in [36] is undertaken by a dedicated data memory management controller (called DMM) designed to efficiently support per flow queuing, providing tens of gigabits per second throughput to an external buffer based on DDR-DRAM technology and many complex operations on these packet queues. The classification of packets into flows is considered part of the protocol processing accomplished prior to packet buffering by a specific processing module denoted as packet classifier. The overall sub-system architecture considered in [36] for packet classification, per-flow queuing and scheduling is shown in Figure 12.19.

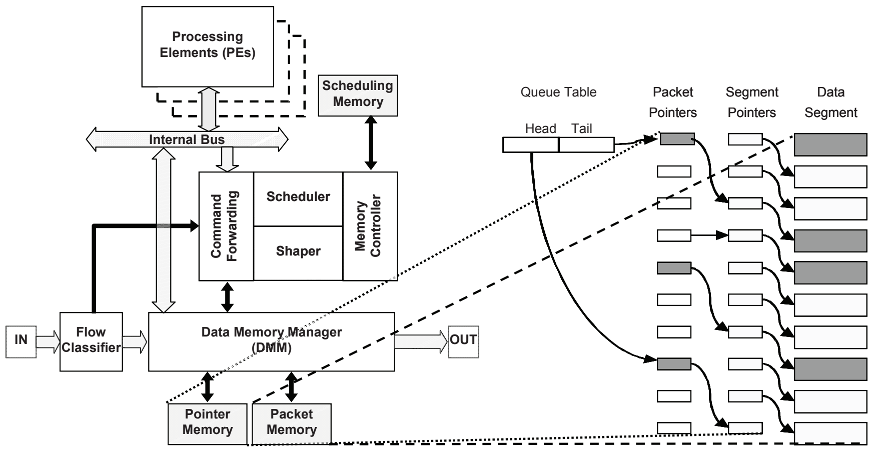

FIGURE 12.19: Packet buffer manager on a system-on-chip architecture.

The main function of the DMM is to store the incoming traffic to the data memory, retrieve parts of the stored packets and forward them to the internal processing elements (PEs) for protocol processing. The DMM is also responsible to forward the stored traffic to the output, based on a programmable traffic-shaping pattern. The specific design reported in [36] supports two incoming and two outgoing data paths at 2.5 Gbps line rate each; there is one for receiving traffic from the network (input), one for transmitting traffic to the network (output), and one bi-directional for receiving and sending traffic from/to the internal bus. It performs per flow queuing for up to 512K flows.

The DMM operates both at fixed length or variable length data items. It uses DRAM for data storage and SRAM for segment and packet pointers. Thus, all manipulations of data structures occur in parallel with data transfers, keeping DRAM accesses to a minimum. The architecture of the DMM is shown in Figure 12.20. It consists of five main blocks: the data queue manager (DQM), data memory controller (DMC), internal scheduler, segmentation block and reassembly block. Each block is designed in a pipeline fashion to exploit parallelism and increase performance. In order to achieve efficient memory management in hardware, the incoming data items are partitioned into fixed size segments of 64 bytes each. Then, the segmented packets are stored in the data memory, which is segment aligned. Segmentation and reassembly blocks perform this function. The internal scheduler forwards the incoming commands from the four ports to the DQM, giving different service priorities to each port. The data queue manager organizes the incoming packets into queues. It handles and updates the data structures, kept in the pointer memory. The data memory controller performs the low level read and writes to the data memory minimizing bank conflicts in order to maximize DRAM throughput as described below.

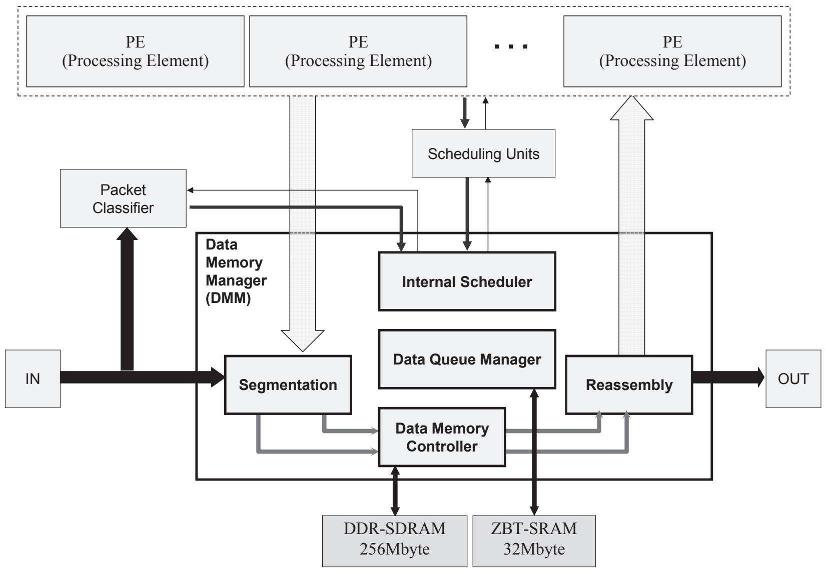

FIGURE 12.20: DMM architecture.

The DMM reported in [36] provides a set of commands in order to support the diverse protocol processing requirements of any device handling queues. Beyond the primitive commands of "enqueue" and "dequeue", the DMM features a large set of 18 commands to perform various manipulations on its data structures (a list of the commands is given in Table 12.3 in Section 12.5.2.2

along with the performance measured for the execution of these commands). Thus it can be incorporated in any embedded system that should handle queues.

DDR-DRAM has been chosen for the data memory because it provides adequate throughput and large storage space for the 512K supported queues, at a low cost as already discussed above. The DDR-SDRAM module used has a 64-pin data bus, which runs at 133 MHz clock frequency, providing 17.024 Gbps total throughput. The large number of the required pointer memory accesses requires a high throughput low latency pointer memory. SRAM has been selected as pointer memory, which provides the required performance. Typical SRAMs working at 133 MHz clock frequency provide 133M accesses per second or about 8.5 Gb/sec.

The data memory space is organized into fixed-size buffers (named segments), which is a usual technique in all memory management implementations. The length of segments is set to 64 bytes because this size minimizes fragmentation loss. For each segment, in the data memory, a segment pointer and a packet pointer are assigned. The addresses of the data segments and the corresponding pointers are aligned, as shown in Figure 12.19, in the sense that a data segment is indexed by the same address as its corresponding pointer. For example, the packet and segment pointers of the segment 0 are in the address 0 in the pointer memory.

The data queues are maintained as single-linked lists of segments that can be traversed from head to tail. Thus, head and tail pointers are stored per queue on a queue table. Head pointers point to the first segment of the head packet in the queue, while the tail pointer indicates the first segment of the tail packet. The DMM can handle traffic at variable length objects (i.e., packets) as well as at fixed-size data items. This is achieved by using two linked lists per flow: one per segment and one per packet. Each entry in the segment level list stores the pointer that indicates the next entry in the list. The maximum number of entries within a data queue equals the maximum number of segments the data memory supports. The packet pointer field has the valid bit set only in the entry that corresponds to the first segment of a packet. The packet pointer also indicates the address of the last segment of a packet. The number of entries of the packet lists is lower than the number of entries of the corresponding segment lists in a typical situation. However, in the worst case the maximum number of entries in the packet level lists is equal to the number of segment level lists, which equals the maximum number of the supported segments in the data memory.

Supporting two types of queues (packet, segment) requires two free lists, one per type. This results in double accesses for allocating and releasing pointers. The above flexible data structures, minimize memory accesses and can support the worst-case scenarios. The two types of linked lists are identical and aligned. In other words, there is only one linked list with two fields: segment and packet pointers. Segment pointers indicate the next segment in the list.

12.5.2.1 Optimization Techniques

The following subsections describe the optimization techniques used in the design to increase performance and reduce the cost of the system.

- **Free List Organization.** The DRAM provides high throughput and capacity at the cost of high latency and throughput limitations due to bank conflicts. A bank conflict occurs when successive accesses address the same bank, and in such case the second access must be delayed until the bank is available Bank conflicts reduce data memory throughput utilization. Hence, special care must be given to the buffer allocation and deallocation process. In [18] there is a proof of how, by using a single free list, a user can minimize the memory accesses during buffer releasing (i.e., delete or dequeue of a large size packet requires O(1) accesses to the pointer memory). However, this scheme increases the possibility of a bank conflict during an enqueue operation. On the other hand, using one free list per memory bank (total of eight banks in the current DRAM chips) minimizes or even avoids bank conflicts during enqueueing but increases the number of memory accesses during packet dequeueing-deletion to O(N). A trade-off of these two schemes, which minimizes the memory accesses and bank conflicts, is to use two free lists and allocate buffers for packet storing from the same free list. Additionally, the support of page-based addresses on the DRAM results in reduction up to 70 percent in the number of bank conflicts during writes and 46 percent during reads.

- **Memory Access Reordering.** The execution of an incoming operation, such as enqueue, dequeue, delete or append packet, sends read and write commands to the pointer memory to update the corresponding data structures. Successive accesses may be dependent. Due to access dependencies, the latency to execute an operation is increased. By reordering the accesses in an effective manner, the execution latency is minimized and thus the system performance increased. This reordering is performed for every operation and was measured to achieve a 30 percent reduction in the access latency.

- **Memory Access Arbitration.** Using the described free list organization, the write accesses to the data memory can be controlled to minimize the bank conflicts. Similar control cannot be performed to read accesses because they are random and unpredictable. Thus, a special memory access arbiter is used in the data memory controller block to shape the flow of read and write accesses to avoid bank conflicts. Memory accesses are classified in four FIFOs (one FIFO per port). The arbiter implements a round-robin policy. It selects an access only if it belongs to a non-busy bank. The information for bank availability is achieved by keeping the data memory access history (last three accesses). This function can reduce bank conflicts by 23 percent. It also reduces the hardware complexity of the DDR memory controller.

- **Internal Backpressure.** The data memory manager uses internal backpressure to delay incoming operations that correspond to blocked flows or blocked devices. The DMM keeps data FIFOs per output port. As soon as these FIFOs are about to overflow, alarm backpressure signals are asserted to suspend the flow of incoming operations related to this blocked datapath. Internal backpressure avoids overflows and data loss. This technique achieves DDM engine architecture reliability by using simple hardware.

12.5.2.2 Performance Evaluation of Hardware Queue Management Engine

Experiments on the DMM design were performed with the support of software and micro-code specifically developed for an IP packet filtering application executed on the embedded micro-engines of the PRO3 NPU presented in [37].

TABLE 12.3: Packet Command and Segment Command Pointer Manipulation Latency

Packet Command	Segment Command	Clock Cycles (5 ns)	Pointer Memory Accesses r: Read; w: Write
Enqueue	Enqueue	10	4r4w
Read	Read	10	3r
Dequeue	Read_N	10	3r
Append	Dequeue_N	13	Min 5 (3r2w)
			Max 8 (3r5w)
Ignore	Overwrite	10	3r
Delete	Overwrite_Segment_length	7	2r1w
Ignore+Delete	Dequeue	13	Min 5 (3r2w)
			Max 8 (3r5w)
	Ignore	4	0
	Ignore+Overwrite_Segment_length	7	2r1w
	Overwrite_Segment_length+Append	11	6r4w
	Overwrite_Segment+Append	11	6r3w

In Table 12.3, the commands supported by the DMM engine are listed. Note that the packet commands are internally translated into segment commands and only segment commands are executed at the low level controller. Table 12.3 also shows the measured latency of these commands when executing the pointer manipulation functions. The actual data access at the data memory can be done almost in parallel with the pointer handling. In particular, the data access can start after the first pointer memory access of each command has been completed. This is achieved because the pointer memory accesses of each command have been scheduled so that the first one provides the data memory address. Hence, DMM can always handle a queue instruction within 65 ns. Since the data memory is accessed at about 50-60 ns (at the average case), and the major part of the queue handling is done in parallel with the data access, the above DMM engine introduces a minimum latency on the whole system. In other words, in terms of latency, you get the queue handling almost "for free", since the DMM latency is about the same as that of a typical (support only read and write) DRAM subsystem.

Table 12.4 depicts the performance results measured after stressing the DMM with real TCP traffic plugged to the NPU ingress interface (supporting one 2.5 Gbps ingress and one 2.5 Gbps egress interface). This table demonstrates the performance of the DMM in terms of both bandwidth and number of instructions serviced. It also presents the memory bandwidth required by our design to provide the performance specified.

TABLE 12.4: Performance of DMM

Number of Flows	AVG packet size (bytes)	MOperations/s serviced	Pointer Memory BW (Gb/s)	DMM BW (Gb/s)
2	100	8.22	4.53	7.60
2	90	10.08	4.45	9.72
2	128	11.26	4.40	9.20
4	128	10.05	3.70	8.40
4	128	9.44	3.80	9.20
Single	64	10.47	2.68	5.32
Single	64	13.70	3.74	7.04
Single	64	15.43	4.50	9.52
Single	50	13.43	4.42	6.88

Since the DMM in the case of the above 2.5 Gbps NPU should actually service each packet four times, the maximum aggregate throughput serviced by it is 10 Gb/sec. From the results of Table 12.4 it can easily be derived that the worst case is when there is only one incoming flow, which consists of very small packets. This worst case can still be served by the DMM engine operating at 200 Mhz while at the same time having a very large number of idle cycles (more than 25 percent even in the worst case). As described above, a simple DRAM can provide up to 17 Gb/sec of real bandwidth while the SRAM up to 8.5 Gb/sec. The maximum memory bandwidth utilization figures show that even in the worst case scenario the bandwidth required by the DRAM is up to about 14 Gb/sec (equal to the DMM bandwidth plus the measured 37 percent overhead due to bank conflicts and fragmentation) and that of the SRAM is 4.5 Gb/sec. As the internal hardware of the DMM in any of these cases is idle for more than 30 percent of the time, the specific DMM engine design could provide even a sustained bandwidth of 12 Gb/sec.

12.6 Scheduling Engines

Scheduling in general is the task of regulating the start and end times of events that contend for the same resource, which is shared in a time division multiplexing (TDM) fashion. Process scheduling is found as a major function of operating systems that control multi-threaded/multiprocessing computer systems ([31], [16], [5]. Packet scheduling is found in modern data networks as a means of guaranteeing the timely delivery of data with strict delay requirements, hence guaranteeing acceptable QoS to real-time applications and fair distribution of link resources among flows and users. Scheduling in a network processor environment is required either to resolve contention for processing resources in a fair manner (task scheduling), or to distribute in time the transmission of packets/cells (in a network medium) due to traffic management rules (traffic scheduling and/or shaping).

Although electronic technology advances rapidly, all of the NPU architectures discussed above are able to perform protocol processing at wire speed only on a long observation window, imposing buffering needs prior to processing. In the context of network processing described in this chapter, when applying complex processing at the processing elements (PEs), a long latency is inadvertently introduced. In order to efficiently utilize the processing capabilities of the node without causing QoS deterioration to packets from critical applications, an efficient queuing and scheduling mechanism (we will use the term *task scheduling* hereafter) for the regulation of the sequence of events related to the processing of the buffered packets is required. An additional implication stems from the multiprocessing architectures, which are most times employed to achieve the required performance that cannot be achieved by a single processing unit. This introduces an additional consideration in the scheduler design with respect to the maintenance of coherent protocol processing to cope with pipelined or parallel processing techniques, which are also very common.

In the outgoing path of the dataflow through the network processing elements, the transmission profile of the traffic leaving the nodes needs appropriate shaping, to achieve the expected delay and jitter levels. Since the internal scheduling and processing may have altered the temporal traffic properties (i.e., delaying some packets more than others, causing the so-called jitter or burstiness in the traffic profile), or because an application requirement to implement rate control for ingress traffic by a traffic scheduler or shaper appropriately adjusting the temporal profile of packet transmission (called hereafter *traffic scheduler*) is imposed.

12.6.1 Data Structures in Scheduling Architectures

In this section we will describe the basic building blocks of scheduling entities able to support both fixed and variable size packets, and to operate at high speeds, consuming few hardware resources. Such functional entities are frequently found as specialized micro-engines in several NPU architectures, or can be the basic functional elements of specialized NPUs designed to implement complex scheduling of packets across many thousands of packet flows and across many network ports.

The algorithmic complexity of proportionate time sharing solutions is based on per-packet time interval calculations and marking/time-stamping. Such algorithms are applicable only for scheduling tasks that have a predetermined completion time and increased complexity. Many studies have focused on analyzing the trade-offs between accurate implementation of algorithms theoretically shown to achieve fair scheduling among flows and simplified time representation and computations along with aggregate traffic handling to reduce memory requirements related to the handling of many thousands of queues. The simplest scheme for service differentiation is based on serving in simple FIFO order flows, classified based on the destination and/or priority of the corresponding traffic. This service discipline can be applied to traffic with different destinations (output ports or processing units), through the instatiation of multiple FIFOs, to avoid head of line (HOL) blocking.

A frequently employed technique that can reduce complexity and increase the throughput of the scheduler implementation with insignificant performance degradation is the grouping of flows with similar requirements into scheduling queues (SQ). Therefore, while a large number of actual data queues (DQs) can be managed by the queue manager, only a limited number of SQs need to be managed by the scheduler, greatly reducing memory requirements. In the simplest case such grouping can be used to implement a strict prioritized service, i.e., highest priority FIFOs are always serviced first until they become empty. This may also lead to starvation of lower priority queues. In order to avoid the starvation problem, queues need to be served in a cyclic fashion. In the simplest case flows within the same priority group are serviced in round-robin (RR) fashion as in [21]. A more general extension of the above approach results in a weighted round-robin service among NS flow groups (SQs) with proportional service (possibly extended in hierarchical hybrid schemes, e.g., implementing strict priorities between super groups): In this case the flows of the same priority are grouped in NS queues, which are served in a weighted round-robin manner, following an organization similar to that described in [42].

The rationale beyond grouping is to save and move implementation resources from detailed flow information to more elaborate resource allocation mechanisms and to improve overall performance applying the proper classification scheme. Assuming that an information entry is kept per schedulable entity, grouping a number of NF flows to a number NS of scheduling queues, a (NF-NS) reduction in storage requirements is achieved. The economy on memory resources regarding the number of pointers is significant since only 2* NS pointers for the management of the NS priority queues are required (and can be stored on-chip), plus NF next (flow) pointers. Flows are grouped to scheduling queues according to some classification rule which depends on the application/configuration. Although the mapping of flows to scheduling queues requires some information maintenance, it requires much less than the size of saved information. Apart from the memory space requirement reduction, reducing the number of schedulable entities facilitates a high decision rate in general, which proves to be mandatory for high-speed network applications.

12.6.2 Task Scheduling

In NPUs, the datapath through the system originates at the network interface, where packets are received and buffered in the input queue. Considering a parallel implementation of multiple processing elements (PEs) the role of the task scheduler is to assign a packet to each of the PEs whenever one of them becomes idle, after verifying that the packet/flow is eligible for service. This latter eligibility check of a packet from a specific flow before forwarding to a processor core (PE) is mandatory in order to maintain the so-called processor consistency. A multiprocessor is said to be processor consistent if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program. To do this effectively, the scheduler can pick any of the packets in the selection buffer, cross-checking a state table indicating the availability of PEs as well as potential state-dependencies of a specific flow (e.g., packets from the same flow may not be forwarded to an idle processor if another packet is already under processing in one of the PEs in order to avoid state dependencies). A packet removed from the selection buffer for processing is replaced by the next packet from the input queue. Processed packets are placed into an output queue and sent to the outgoing link or the switch fabric of a router.

Hardware structures for the efficient support of load balancing of traffic in such multiprocessor SoCs in very high speed applications have appeared only recently. The assumptions and application requirements in these cases differ significantly from the processing requirements and programming models of high-speed network processing systems. NPUs represent a typical multiprocessing system where parallel processing calls for efficient internal resource management. However, the network processor architectures usually follow the run-to-completion mode, distributing processing tasks (which actually represent packet and protocol processing functions) on multiple embedded processing cores, rather than following complex thread parallelism. Load balancing has also been studied in [22]. The analysis included in [22] followed the assumption of multiple network processors operating in parallel with no communication and state sharing capabilities between them, as well as the requirement to minimize re-assignments of flows to different units for this reason as well as to avoid packet reordering. These assumptions are relaxed when the processing units are embedded on a single system-on-chip and access to shared memories as well as communication and state locking mechanisms are feasible. Current approaches for processor sharing in commercial NPUs are discussed below.

The IBM PowerNP NP4GS3 [3] includes eight dyadic protocol processing units (DPPUs) and each one contains two core language processors (CLPs). Sixteen threads at most can be active, even though each DPPU can support up to four processes. The dispatch event controller (DEC) schedules the dispatch of work to a thread, and is able to load balance threads on the available DPPUs and CLPs, while a completion unit detects their state and maintains frame order within communication flows. However, it can process 4.5 million packets per second layer 2 and layer 3 switching, while operating at 133 MHz.

The IXP 2800 network processor [19] embeds 16 programmable multi-threaded micro-engines that utilize super-pipeline technology that allows the forwarding of data items between neighboring micro-engines. Hence, the processing is based on a high-speed pipeline mechanism rather than on associating one micro-engine to the processing of a full packet (although this latter case is possible via its local dispatchers).

The Porthos network processor [32] uses 256 threads in 8 processing engines. In this case, in-order packet processing is controlled mainly by software, and it is assisted by a hardware mechanism which tags each packet with a sequence number. However, load balancing capability is limited and completely controlled by software.

12.6.2.1 Load Balancing

An example of such an on-chip core has been presented in [25]. The scheduler/load balancer presented in [25] is designed to allocate the processing resources to the different packet flows in a fair (i.e., weighted) manner according to pre-configured priorities/weights, whereas the load balancing mechanism supports efficient dispatching of the scheduled packets to the appropriate destination among a set of embedded PEs. The main datapath of the NPU in this case is the one examined in previous chapters and is shown in Figure 12.21.

This set of PEs shown in Figure 12.21 may be considered to be of similar capacity and characteristics (so effectively there is only one set) or may be differentiated to independent sets. In any case each flow is assigned to such a set. A load-balancing algorithm is essential to evenly distribute packets to all the processing modules. The main goal for the load balancing algorithm is that the workloads of all the processing elements are balanced and throughput degradation is prevented. Implementation of load balancing in this scheme is done in two steps. First, based on the results of the classification stage, packet flows undergoing the same processing (e.g., packets from similar interfaces using the same framing and protocol stacks and enabled services use a predefined set of dedicated queues) are distributed among several queues, which

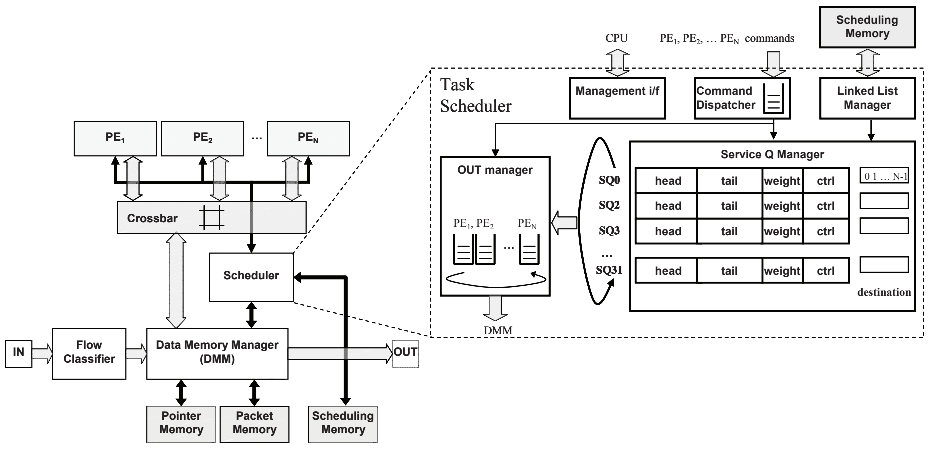

FIGURE 12.21: Details of internal task scheduler of NPU architecture described in [25].

represent the set of internal flows that are eligible for parallel processing. In the second step, the task scheduler based on the information regarding the traffic load of these queues (i.e., packet arrival events) selects and forwards packets based on an appropriate service policy. The application software that is executed by a PE on a given packet is implicitly defined by the flow/queue to which the packet belongs.

Pre-scheduled flows are load-balanced to the available PEs by means of a strict service mechanism. The scheme described in [25] is based on the implementation of an aging mechanism used in the core of a crossbar switch fabric (this on-chip interconnection architecture is usually called network-on-chip, NoC). The reference crossbar switch used is based on a traditional implementation of a shared-memory switch core; all target ports corresponding to a programmable processing core (PE) access this common resource with the aid of a simple arbitrating mechanism.

A block diagram of the load-balancing core described in [25] is shown in Figure 12.22. The main hardware data structures used (assuming 64 available on-chip PEs and 1M flows) are the following:

- The free list maintains the occupancy status of the rest tables; any PE may select any waiting flow residing in the flow memory.
- The FLOW memory records the flow identifier of a packet waiting to be processed.
- The DEST memory stores a mask denoting the PEs that can process this packet (i.e., can execute the required application code).
- The AGE memory stores the virtual time (in the form of a bit vector)

based on the arrival time of this packet (it is called virtual, because it represents the relative order among the scheduled flows).

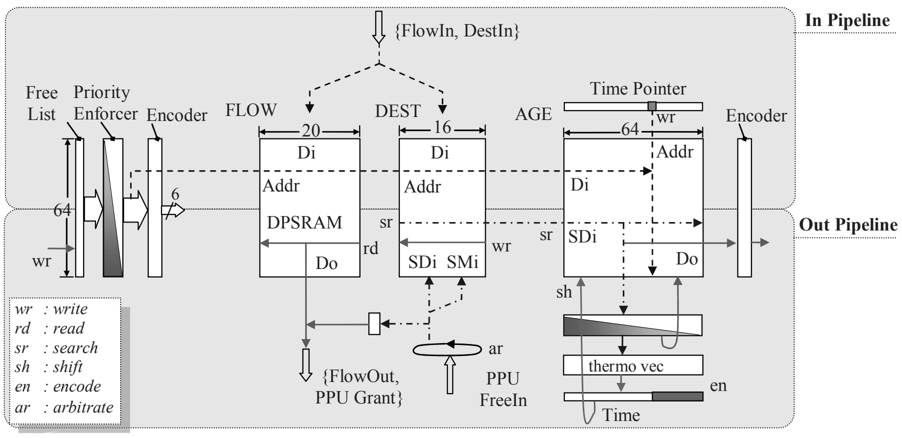

FIGURE 12.22: Load balancing core implementation [25].

Two pipeline operations are executed in parallel for incoming packets and packets that have completed processing at the PEs. The in pipeline is triggered if no PE service is in progress. It is responsible to store the flow identifier in an available slot provided by the free list, and mark the corresponding destination mask in the DEST table. The virtual time indicator is updated and stored in the AGE table, aligned with the flow identifier and the destination mask, which is used to guarantee service according to the arrival times of the requests. Finally, a filter/aggregation mask is updated to indicate the PEs needed to serve the waiting flows. After applying the filter mask to the PE availability vector indicator, the DEST table is searched to discover the flows scheduled to this PE. This is a "don't care" search, since the matching flows could be load-balanced to other PEs as well. In order to serve the oldest flow, the AGE table is searched in the next stage, based on the previous outcome. The outcome of this searching is: a) to read the AGE table and produce the winner flow to be served, and b) to shift all the younger flows to the right and automatically erase the age of the winner. The flow memory is read after encoding the previous outcome, and finally, the free list and DEST tables are updated accordingly. The basic structures used include multi-ported memories, priority enforcers, a content addressable memory (CAM) that allows ternary search operations (for the implementation of the DEST table). The most complex data structure is the AGE block. This is also a CAM, which differs in that it performs exact matches. Additionally, it has a separate read/write port and supports a special shift operation; it shifts each column vector to the right when indicated by a one in a supplied external mask. The circuit thermo vec performs a thermometer decoding. It transforms the winner vector produced by the priority enforcer to a sequence of ones from the located bit position until the left most significant bit. This is "ANDed" with the virtual time vector to produce the shift enable vector. Thus, only the active columns are shifted.

The concept of operation described above is also similarly found in the case of the Porthos NPU ([32]) which uses 256 threads in eight PEs (called tribes in [32]). In this case, in-order packet processing is controlled mainly by software, and it is assisted by a hardware mechanism which tags each packet with a sequence number. However, load balancing capability is limited and completely controlled by software. The hardware resources supporting this functionality are based on the interconnection architecture of the Porthos NPU shown in Figure 12.23.

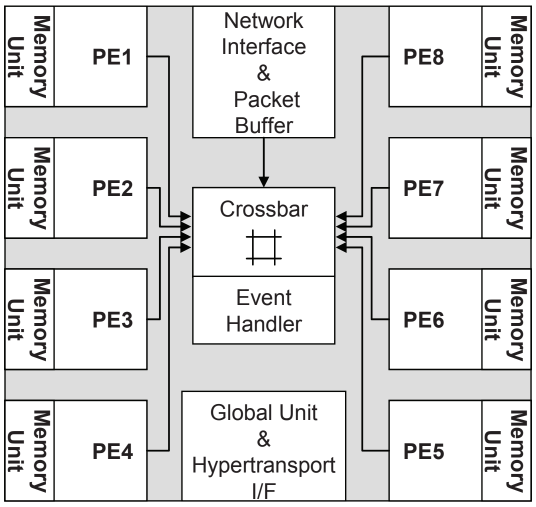

FIGURE 12.23: The Porthos NPU interconnection architecture [32].

The Porthos interconnect block consists of three modules: event handling, arbiter and crossbar (comprising 10 input and 8 output 64-bit wide ports supporting backpressure for busy destinations). The event module collects event information and activates a new task to process the event. It spawns a new packet processing task in one of the PEs via the interconnect logic based on external and timer (maskable) interrupts. There are two (configurable) methods to which an interrupt can be directed. In the first method, the interrupt is directed to any PE that is not empty. This is accomplished by the event module making requests to all eight destination PEs. When there is a grant to one PE, the event module stops making requests to the other PEs and starts the interrupt handling process. In the second method, the interrupt is directed to a particular PE.

The arbiter module performs arbitration between sources (PEs, network block, event handling module and transient buffers) and destinations. The arbiter needs to match the source to the destination in such a way as to maximize utilization of the interconnect, while also preventing starvation using a round-robin prioritizing scheme. This matching is performed in two stages. The first stage selects one non-busy source for a given non-busy destination. The second stage resolves cases where the same source was selected for multiple destinations. The arbitration scheme implemented in Porthos is "greedy", meaning it attempts to pick the requests that can proceed, skipping over sources and destinations that are busy. In other words, when a connection is set up between a source and a destination, the source and destination are locked out from later arbitration. With this scheme, there are cases when the arbiter starves certain contexts. It could happen that two repeated requests, with overlapping transaction times, can prevent other requests from being processed. To prevent this, the arbitration may operate in an alternative mode. In the non-greedy mode for each request that cannot proceed, there is a counter that keeps track of the number of times that request has been skipped. When the counter reaches a configurable threshold, the arbitration will not skip over this request, but rather wait at the request until the source and destination become available. If multiple requests reach this priority for the same source or destination, one-by-one they will be allowed to proceed in a strict round-robin fashion.

An architectural characteristic of the Porthos NPU is the support for the so-called task migration from one PE to another, i.e., a thread executing on a PE transferred to another. When migration occurs, a variable amount of context follows the thread. Specific thread migration instructions are supported specifying a register that contains the destination address and an immediate response that contains the amount of thread context to preserve. Thread migration is a key feature of Porthos, providing support to the overall loosely coupled processor/memory architecture. Since the memory map is not overlapped, every thread running in each PE has access to all memory. Thus, from the standpoint of software correctness, migration is not strictly required. However, the ability to move the context and the processing to an engine that is closer to the state that the processing is operating on allows a flexible topology to be implemented. A given packet may do all of its processing in a specific PE, or may follow a sequence of PEs (this decision potentially is made on a packet-by-packet basis). Since a deadlock can occur when the PEs in migration loops are all full, Porthos uses two transient buffers in the interconnect block to break such deadlocks, with each buffer capable of storing an entire migration (66 bits times maximum migration cycles). These buffers can be used to transfer a migration until the deadlock is resolved by means of atomic software operations at a cost of an additional delay.

12.6.3 Traffic Scheduling

The Internet and the associated TCP/IP protocols were initially designed to provide best-effort (BE) service to end users and do not make any service quality commitment. However, most multimedia applications are sensitive to available bandwidth and delay experienced in the network. To satisfy these requirements, two frameworks have been proposed by IETF: the integrated services (IntServ), and the differentiated services (DiffServ) [47], [26]. The IntServ model provides per-flow QoS guarantee and RSVP (resource reservation protocol) is suggested for resource allocation and admission control. However, the processing load is too heavy for backbone routers to maintain state of thousands of flows. DiffServ is designed to scale to large networks and gives a class-based solution to support relative QoS. The main idea of DiffServ is to minimize state and per-flow information in core routers by placing all packets in fairly broad classes at the edge of network. The key ideas of DiffServ are to: (a) classify traffic at the boundaries of a network, and (b) condition this traffic at the boundaries. Core devices perform differentiated aggregate treatment of these classes based on the classification performed by the edge devices. Since it is highly scalable and relatively simple, DiffServ may dominate the next generation Internet in the near future. Its implementation in the context of a network routing/switching node is shown in Figure 12.24.

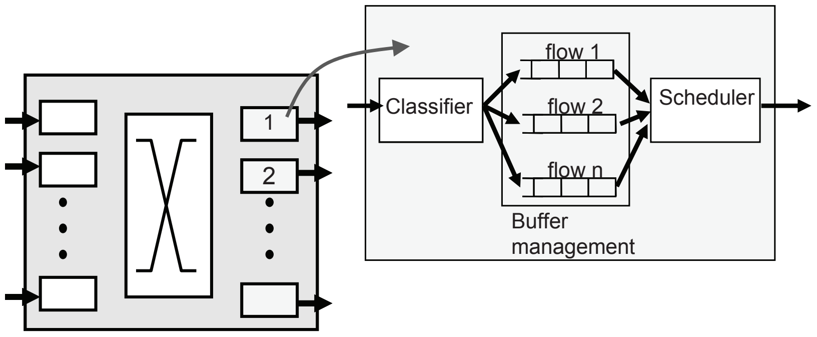

FIGURE 12.24: Scheduling in context of processing path of network routing/switching nodes.

In DiffServ, queues are used for a number of purposes. In essence, they are only places to store traffic until it is transmitted. However, when several queues are used simultaneously in a queueing system, they can also achieve effects beyond those for given traffic streams. They can be used to limit variation in delay or impose a maximum rate (shaping), to permit several streams to share a link in a semi-predictable fashion (load sharing) or move variation in delay from some streams to other streams. Queue scheduling schemes can be divided into two types: work-conserving and non-work-conserving. A policy is work-conserving if the server is never idle when packets are backlogged. Among work-conserving schemes, fair queueing is the most important category. WFQ (weighted fair queueing) [38], WF2Q, WF2Q+ [4] and all other GPS-based queueing algorithms belong to fair queueing. Another important type of work-conserving is the service curve scheme, such as SCED [8] and H-FSC [43]. The operation of these algorithms is schematically described in Figure 12.25.

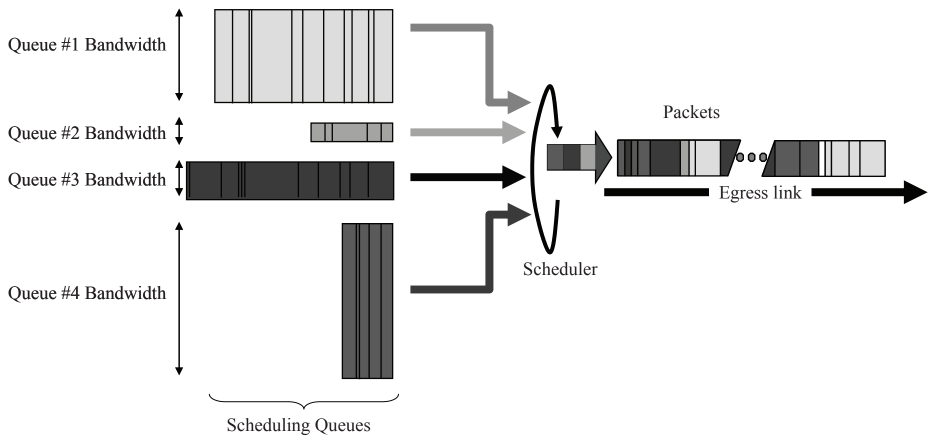

FIGURE 12.25: Weighted scheduling of flows/queues contending for same egress network port.

All these schemes present the traffic distortion [29] problem and traffic characterization at the entrance of the network would not be valid inside the network. And traffic can get more bursty in the worst case. In downstream switches, more buffer spaces are required to handle traffic burstiness and the receiver also needs more buffer space to remove jitter. Non-work-conserving schemes (also called shapers) are proposed in order to control traffic distortion inside a network. A policy is non-work-conserving if the server may be idle even when packets are backlogged. From the definition we can see that non-work-conserving schemes allow the output link to be idle even when there are packets waiting for service in order to maintain the traffic pattern. So bandwidth utilization ratio may be not be high in some cases.

The design of weighted schedulers can follow the generic architecture described above for task scheduling to implement multiple traffic management mechanisms in an efficient way. An extension of the NPU architecture that could exploit these traffic management extensions is shown in Figure 12.26(a). This architecture suits better the needs of multi-service network elements found in access and edge devices that act as traffic concentrators and protocol gateways. This architecture represents a gateway-on-chip paradigm exploiting the advances in VLSI technology and SoC design methodologies that enable the easy integration of multiple IP cores on complex designs. In cases like this the queuing and scheduling requirements are complicated. Apart from the high number of network flows and classes of service (CoS) that need to be supported, another hierarchy level is introduced that necessitates the extension of the scheduler architecture described above to support multiple virtual and physical output interfaces as shown in Figure 12.26(b).

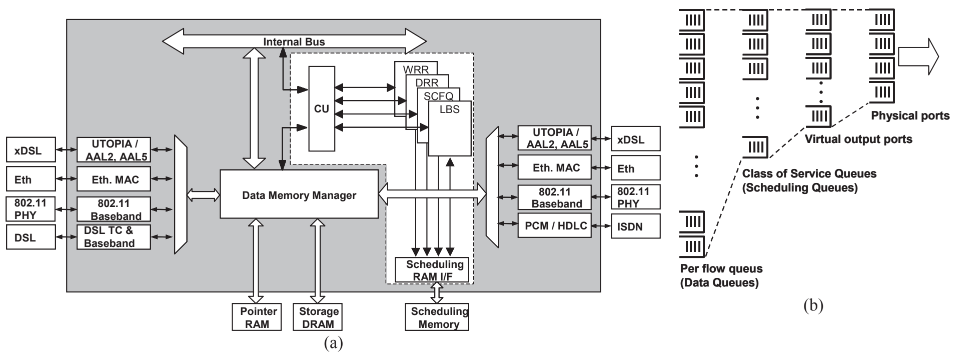

FIGURE 12.26: (a) Architecture extensions for programmable service disciplines. (b) Queuing requirements for multiple port support.

The generic scheduler architecture, as described in Figure 12.21, and following the organization presented in Figure 12.26 (a) and (b) which incorporates the internal to the NPU task scheduler inherently supports these hierarchical queuing requirements by means of independent scheduling per DQ, SQ and destination (port). Furthermore, the same module can implement different service disciplines (like WRR and DRR) in a programmable fashion with the same hardware resources. Thus, by proper organization of flows under SQs per CoS, efficient virtual and physical port scheduling can also be achieved as described in [35]. Implementation of more scheduling disciplines can also be achieved easily, by simply adding the service execution logic (finite state machine or FSM) as a co-processing engine, since the implementation area is small and operation and configuration is independent among them. Even a large number of schedulers could be integrated at low cost. Apart from the implementation of additional FSMs and potentially the associated on-chip memory (although insignificant) the only hardware extension required is the extension of the arbiter and memory controller modules to support a larger number of ports. The required throughput of the pointer memories used remains the same as long as the aggregate bandwidth of the incoming network interfaces is at most equal to the throughput offered by the DMM unit. The only limitation is related to the number of supported SQs, which represent one CoS queue each. Thus, the number of independently scheduled classes of service is directly proportional to the hardware resources that will be allocated for the implementation of the SQ memories and priority enforcers for fast searching in these memories, which can be extended to very high numbers of SQs as presented in [24]. In addition, functionality already present in the current scheduler implementation allows for deferring service of one SQ and manipulation of its parameters under software control. This feature offers itself for easy migration of one CoS from one scheduling discipline to another in this extended architecture.

With these extensions the NPU can efficiently support concurrent scheduling mechanisms for network traffic, crossing even dissimilar interfaces. Scheduling of variable length packet flows having as destinations packet interfaces (like Ethernet, packet-over-SONET etc.) can be scheduled by means of a packet scheduling algorithm like DRR or self-clocked fair queueing (SCFQ, [12]). The efficient implementation of packet fair queuing algorithms like SCFQ, according to the generic methodology presented in this section has also been discussed in [39]. Moreover, a novel feature of this architecture is its flexibility to implement hierarchical scheduling schemes only with pointer movement without necessitating data movement. Scheduling packets over multiple interfaces of the same type (e.g., multiple Ethernet interfaces) is easily achieved by assigning appropriate weights (that represent the relative share of a flow with respect to the aggregate capacity of the physical links) and different destinations (port) per flow. The only remaining hardware issue that requires attention is the handling of busy indication signals from the different physical ports to determine schedulable flows/SQs.

12.7 Conclusions

State-of-the-art telecommunication systems require modules with increased throughput in terms of packets processed per second and with advanced functionality extending to multiple layers of the protocol stack. High-speed datapath functions can be accelerated by hard wired implementations integrated as processing cores in multi-core embedded system architectures. This allows each core to be optimized either for processing intensive functions to alleviate bottlenecks in protocol processing or intelligent memory management techniques to sustain the throughput for data and control information storage and retrieval and exceed the performance of legacy SW-based implementations on generic microprocessor based architectures, which cannot scale to gigabit-per-second link rates.

The network processing units (NPUs) that we examined in this chapter in the strict sense are fully programmable chips like CPUs or DSPs but, instead of being optimized for the task of computing or digital signal processing, they have been optimized for the task of processing packets and cells. In this sense NPUs combine the flexibility of CPUs with the performance of ASICs, accelerating the development cycles of system vendors, forcing down cost, and creating opportunities for third-party embedded software developers.

NPUs in the broad sense encompass both dedicated and programmable solutions:

- Dedicated line-aggregation devices that combine several channels of high-level data link control support sometimes optimized for a specific access system such as DSL
- Intelligent memories, e.g., content-addressable memories that support efficient searching of complex multi-dimensional lookup-tables
- Application-specific ICs optimized for one specific protocol processing task, e.g., encryption
- Programmable processors optimized for one specific protocol processing task, e.g., header parsing
- Programmable processors optimized for several protocol processing tasks

The recent wave of network processors is aimed at packet parsing and header analysis. Two evolutions favor programmable implementations. First, the need to investigate and examine more header fields covering different layers of the OSI model, make an ASIC implementation increasingly complex. Secondly, flexibility is required to deal with emerging solutions for QoS and other services that are not yet standardized. The challenge for the programmable network processors lies in the scalability to core applications running at 10 Gbits/s and above (which is why general-purpose processors are not up to the job).

The following features of network processors have been taken into account to structure this case study:

- Target application domain (LAN, access, WAN, core/edge etc.).
- Target function (data link processing including error control, framing etc., classification, data stream transformation including encryption, compression etc., traffic management including buffer management, prioritization, scheduling, shaping, and higher layer protocol/FSM implementation)
- Architecture characteristics including:
  - Architecture based on instruction-set processor (ISP), programmable state machine (PSM), ASIC (non-programmable), intelligent memory (CAM)
  - Type of ISP (RISC, DSP)
  - Centralized or distributed architecture
  - Programmable or dedicated
  - DSP acceleration through extra instructions or co-processors

  – Presence of re-configurable hardware

- Software development environment (for programmable NPUs)
- Performance in terms of data rates
- Implementation: processing technology, clock speed, package, etc.

Review Questions and Answers

[Q_1] **What is the range of applications that are usually executed in network processing units?**
Refer to Section 12.1 of the text. Briefly, network processing functions can be summarized as follows:
• Implementation of physical ports, physical layer processing and traffic switching
• Framing
• Classification
• Modification
• Content/protocol processing
• Traffic engineering, scheduling and shaping algorithms

[Q_2] **What are the processing requirements and the bottlenecks that led to the emergence of specialized NPU architectures?**
Two major factors drive the need for NPUs: i) increasing network bit rate, and ii) more sophisticated protocols for implementing multi-service packet-switched networks. NPUs have to address the above communications system performance issues by coping with three major performance-related resources in a typical data communications system:

1. Processing cores (limited processing capability and frequency of operation of single, general purpose, processing units)
2. System bus(es) (limited throughput and scalability)
3. Memory (limited throughput and scalability of centralized architectures)

[Q_3] **What are the main differences in NPU architectures targeting access/metro networks compared to those targeting core networks?**
Due to the different application requirements there are the following differences:
• Overall throughput (access processors usually achieve throughputs in

the order of 1 Gbps, which is adequate for most access network technologies, whereas core networks may require an order of magnitude higher bandwidth i.e., 10-40 Gbps)
• Number of processing cores (single-chip IADs can integrate only a couple of general purpose CPUs, whereas high-end NPUs can integrate 4-64 processing cores)
• Multiplicity and dissimilarity of interfaces/ports (access processors frequently must support bridging between multiple networks of different technologies, whereas core processors are required to interface to high-speed line-cards and switching fabrics through a limited set of standardized interfaces)
• Architectural organization (access processors frequently require custom processing units since intelligent content processing, e.g., (de)encryption, (de)compression, transcoding, content inspection, etc. is usually pushed to the edge of the network, whereas core processors require ultimate throughput and traffic management which is addressed through massively parallel, pipelined and programmable FSM architectures with complicated memory management and scheduling units)

[Q_4] **Why is latency not very important when packet-processing tasks are executed on a network processor? What happens when such a task is stalled?**
Usually a network processor time-shares between many parallel tasks. Typically such tasks are independent, because they process packets from different flows. So, when a task is stalled (e.g., on a slow external memory access or a long-running coprocessor operation) the network processor switches to another waiting task. In this way, almost no processing cycles get wasted.

[Q_5] **Which instruction-level-parallel processor architecture is more area-efficient: superscalar, or VLIW? Why?**
Very long instruction word (VLIW) is more area-efficient than superscalar, because the latter includes a lot of logic dedicated to "discovering" at run time instructions that can be executed simultaneously. VLIW architectures include no such logic; instead they require that the compiler schedules instructions statically at compile time.

[Q_6] **What are the pros and cons of homogeneous and heterogeneous parallel architectures?**
By specializing each processing element to a narrowly defined processing task, heterogeneous architectures can achieve higher peak performance for each individual task. On the other hand, with such architectures, one has to worry about load balancing: the system architects need to choose the correct number and performance of each type of PE, a problem with no general solution, while the users must be careful to code

the applications in a way that balances the load between the different kinds of available PEs. With homogeneous architectures, the architect only needs to replicate a single type or PE as many times as silicon area allows, while the user can always take advantage of all available PEs. This of course comes at the cost of lower peak PE performance.

[Q_7] **Define multi-threading.**
A type of lighweight time-sharing mechanism. Threads are akin to processes in the common operating system sense, but hardware support allows very low (sometimes zero) overhead when switching between threads. So, it is possible to switch to a new thread even when the current thread will be stalled for a few clock cycles (e.g., an external memory access or an operation executed on a coprocessor). This allows the processor to take advantage of (almost) all processing cycles, by making progress on an alternate thread when the current one is stalled even briefly.

[Q_8] **Explain how the PRO3 processor overlaps processing with memory accesses.**
Refer to Section 12.3.2 of the text.

[Q_9] **Mention some types of custom instructions specific to network processing tasks.**
• Extraction of bit fields of arbitrary lengths and from arbitrary offsets within a word
• Insertion of bit fields of arbitrary lengths into arbitrary offsets within a word
• Parallel multi-way branches (or parallel comparisons, as in the IXP architecture)
• CRC/checksum calculation or hash function evaluation

[Q_10] **Define the problem of packet classification**
Refer to the introductory part of Section 12.4 of the text.

[Q_11] **Name a few applications of classification**
• Destination lookup for IP forwarding
• Access control list (ACL) filtering
• QoS policy enforcing
• Stateful packet inspection
• Traffic management (packet discard policies)
• Security-related decisions
• URL filtering
• URL-based switching
• Accounting and billing

[Q_12] **What are the pros and cons of CAM-based classification architectures?**

CAM-based lookups are the fastest and simplest ways to search a rule database. However, CAMs come at high cost and have high power dissipation. In addition, the capacity of a CAM device may enforce a hard upper bound on database size. (Strictly speaking, the same is true for algorithmic architectures, but since these usually rely on low-cost DRAM, it is easier to increase the memory capacity for large rule databases.)

[Q_13] **How does iFlow's Address Processor exploit embedded DRAM technology?**
First of all, it combines the database storage with all the necessary lookup and update logic into one device, thus reducing overall cost. Second, it takes advantage of a very wide internal memory interface to read out many nodes of the data structure and make that many comparisons in parallel, thus improving performance.

[Q_14] **What are the main processing tasks of a queue management unit?**
Refer to the introductory parts of Sections 12.5 and 12.5.2 of the text.

[Q_15] **What are the criteria of selecting memory technology when designing queue management units for NPU?**
Refer to Sections 12.5.1 and 12.5.2 of the text. Briefly, the memory technology of choice should provide:
• Adequate throughput depending on the data transaction (read/write, single/burst etc.) requirements
• Adequate space depending on the storage requirements
• Limited cost, board space and power consumption

[Q_16] **What are the main bottlenecks in queue management applications and how are they addressed in NPU architectures?**
Refer to Sections 12.5.1 and 12.5.2 of the text. Briefly, the main performance penalties are due to timing limitations related to successive memory operations depending on the memory technology. DRAM is an indicative case of memory technology which requires sophisticated controllers due to the limitations in the order of accesses, depending on its organization in banks, its requirements for pre-charging cycles, etc., to enhance its performance and better utilize its resources. Such controllers can enhance memory throughput through multiple techniques e.g., appropriate free list organization, appropriate scheduling of accesses requested by multiple sources enforcing reordering, arbitration, internal backpressure, etc.

[Q_17] **What are the similarities and differences between task and traffic scheduling?**
Both applications are related to resource management based on QoS criteria. In general scheduling refers to the task of ordering in time the

execution of processes, which can either be processing tasks that require the exchange of data inside an NPU or transmission of data packets in a limited capacity physical link. In both cases the data on which the process is going to be executed are ordered in multiple queues served with an appropriate discipline that guarantees some performance criteria (delay, throughput, data loss etc.). Depending on the application, different requirements need to be met, in order delivery, rate-based flow limiting, etc. Task processing has three important differences in the way it should be implemented: i) the finish time of a processing task in contrast to packet transmission delays, which depend only on link capacity and packet length, may be unknown or hard to determine (e.g., due to the stochastic nature of branch executions that depend on the content of data which are not *a priori* known to the scheduler), ii) the availability of the resources varies dynamically and may have specific limitations due to dependencies in pipelined execution or atomic operations in parallel processing, etc., and iii) the optimization of throughput in task scheduling may require load balancing, i.e., distribution of tasks to any available resource whereas traffic scheduling needs to coordinate requests for access to the same predetermined resource (i.e., port/link).

[Q_18] **What are the main processing tasks of a traffic scheduling unit?**
Refer to Section 12.6 of the text. Briefly, traffic scheduling requires the implementation of an appropriate packet queuing scheme (a number of priority queues, possibly hierarchically organized) and the implementation of an appropriate arbitration scheme either in a deterministic manner or in the most complex case computing per packet information (finish times) and sorting appropriately among all packets awaiting service (e.g., DRR, SCFQ, WFQ-like algorithms etc.).

Bibliography

[1] BGP routing tables analysis reports. http://bgp.potaroo.net.

[2] Matthew Adiletta, Mark Rosenbluth, Debra Bernstein, Gilbert Wolrich, and Hugh Wilkinson. The next generation of Intel IXP network processors. *Intel Technology Journal*, 6(3):6–18, 2002.

[3] J.R. Allen, B.M. Bass, C. Basso, R.H. Boivie, J.L. Calvignac, G.T. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, A. Kind, J.F. Logan, M. Peyravian, M.A. Rinaldi, R.K. Sabhikhi, M.S. Siegel, and M. Waldvogel. IBM PowerNP network processor: hardware, software, and applications. *IBM Journal of Research and Development*, 47(2/3):177–193, 2003.

[4] Jon C. R. Bennett and Hui Zhang. Why WFQ is not good enough for integrated services networks. In *Proceedings of NOSSDAV '96*, 1996.

[5] Haiying Cai, Olivier Maquelin, Prasad Kakulavarapu, and Guang R. Gao. Design and evaluation of dynamic load balancing schemes under a fine-grain multithreaded execution model. Technical report, *Proceedings of the Multithreaded Execution Architecture and Compilation Workshop*, 1997.

[6] C-port Corp. C-5 network processor architecture guide, C5NPD0-AG/D, May 2001.

[7] Patrick Crowley, Mark A. Franklin, Haldun Hadimioglu, and Peter Z. Onufryk. *Network Processor Design: Issues and Practices*. Morgan Kaufmann, 2003.

[8] R. L. Cruz. SCED+: efficient management of quality of service guarantees. In *Proceedings of INFOCOM'98*, pages 625–642, 1998.

[9] EZchip. Network processor designs for next-generation networking equipment. http://www.ezchip.com/t_npu_whpaper.htm, 1999.

[10] EZchip. The role of memory in NPU system design. http://www.ezchip.com/t_memory_whpaper.htm, 2003.

[11] V. Fuller and T. Li. Classless inter-domain routing (CIDR): The internet address assignment and aggregation plan. RFC4632.

[12] Jahmalodin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of INFOCOM'94 13th Networking Coference for Global Communications*, volume 2, pages 636–646, 1994.

[13] Network Processing Forum Hardware Working Group. Look-aside (LA-1B) interface implementation agreement, August 4 2004.

[14] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[15] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[16] A. El-Mahdy I. Watson, G. Wright. VLSI architecture using lightweight threads (VAULT): Choosing the instruction set architecture. Technical report, Workshop on Hardware Support for Objects and Microarchitectures for Java, in conjunction with ICCD'99, 1999.

[17] IBM. PowerNP NP4GS3.

[18] Motorola Inc. Q-5 traffic management coprocessor product brief, Q5TMC-PB, December 2003.

[19] Intel. Intel IXP2400, IXP2800 network processors.

[20] ISO/IEC JTC SC25 WG1 N912. Architecture of the residential gateway.

[21] Manolis Katevenis. Fast switching and fair control of congested flow in broadband networks. *IEEE Journal on Selected Areas in Communications*, 5:1315–1326, Oct. 1987.

[22] Manolis Katevenis, Sotirios Sidiropoulos, and Christos Courcoubetis. Weighted round robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communications*, 9, 1991.

[23] Donald E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[24] George Kornaros, Theofanis Orphanoudakis, and Ioannis Papaefstathiou. Active flow identifiers for scalable, qos scheduling. In *Proceedings IEEE International Symposium on Circuits and Systems ISCAS'03*, 2003.

[25] George Kornaros, Theofanis Orphanoudakis, and Nicholas Zervos. An efficient implementation of fair load balancing over multi-CPU SoC architectures. In *Proceedings of Euromicro Symposium on Digital System Design Architectures, Methods and Tools*, 2003.

[26] K. R. Renjish Kumar, A. L. An, and Lillykutty Jacob. The differentiated services (diffserv) architecture, 2001.

[27] V. Kumar, T. Lakshman, and D. Stiliadis. Beyond best-effort: Router architectures for the differentiated services of tomorrow's internet. *IEEE Communications Magazine*, 36:152–164, 1998.

[28] Sridhar Lakshmanamurthy, Kin-Yip Liu, Yim Pun, Larry Huston, and Uday Naik. Network processor performance analysis methodology. *Intel Technology Journal*, 6, 2002.

[29] Wing-Cheong Lau and San-Qi Li. Traffic distortion and inter-source cross-correlation in high-speed integrated networks. *Computer Networks and ISDN Systems*, 29:811–830, 1997.

[30] Panos Lekkas. *Network Processors. Architectures, Protocols, and Platforms.* McGraw-Hill, 2004.

[31] Evangelos Markatos and Thomas Leblanc. Locality-based scheduling in shared-memory multiprocessors. Technical report, *Parallel Computing: Paradigms and Applications*, 1993.

[32] Steve Melvin, Mario Nemirovsky, Enric Musoll, Jeff Huynh, Rodolfo Milito, Hector Urdaneta, Koroush Saraf, and Myers Llp. A massively multithreaded packet processor. In *Proceedings of NP2, Held in conjunction with HPCA-9*, 2003.

[33] Aristides Nikologiannis and Manolis Katevenis. Efficient per-flow queueing in DRAM at OC-192 line rate using out-of-order execution techniques. In *Proceedings of ICC2001*, pages 2048–2052, 2001.

[34] Mike O'Connor and Christopher A. Gomez. The iFlow address processor. *IEEE Micro*, 21(2):16–23, 2001.

[35] Theofanis Orphanoudakis, George Kornaros, Ioannis Papaefstathiou, Hellen-Catherine Leligou, and Stylianos Perissakis. Scheduling components for multi-gigabit network SoCs. In *Proceedings SPIE International Symposium on Microtechnologies for the New Millennium, VLSI Circuits and Systems Conference, Canary Islands*, 2003.

[36] Ioannis Papaefstathiou, George Kornaros, Theofanis Orphanoudakis, Kchristoforos Kachris, and Jacob Mavroidis. Queue management in network processors. In *Design, Automation and Test in Europe (DATE2005)*, 2005.

[37] Ioannis Papaefstathiou, Stylianos Perissakis, Theofanis Orphanoudakis, Nikos Nikolaou, George Kornaros, Nicholas Zervos, George Konstantoulakis, Dionisios Pnevmatikatos, and Kyriakos Vlachos. PRO3: a hybrid NPU architecture. *IEEE Micro*, 24(5):20–33, 2004.

[38] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1:344–357, 1993.

[39] Jennifer Rexford, Flavio Bonomi, Albert Greenberg, and Albert Wong. Scalable architectures for integrated traffic shaping and link scheduling in high-speed ATM switches. *IEEE Journal on Selected Areas in Communications*, 15:938–950, 1997.

[40] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, 2001.

[41] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM Sigcomm*, pages 203–214, September 1998.

[42] Donpaul C. Stephens, Jon C. R. Bennett, and Hui Zhang. Implementing scheduling algorithms in high speed networks. *IEEE JSAC*, 17:1145–1158, 1999.

[43] Ian Stoica, Hui Zhang, and T.S.E Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Transactions on Networking*, 8(2):185–199, 2000.

[44] Sandy Teger and David J. Waks. End-user perspectives on home networking. *IEEE Communications Magazine*, 40:114–119, 2002.

[45] K. Vlachos, T. Orphanoudakis, Y. Papaefstathiou, N. Nikolaou, D. Pnevmatikatos, G. Konstantoulakis, J.A. Sanches-P, and N. Zervos. Design and performance evaluation of a programmable packet processing engine (ppe) suitable for high-speed network processors units. *Microprocessors and Microsystems*, 31(3):188–199, May 2007.

[46] David Whelihan and Herman Schmit. Memory optimization in single chip network switch fabrics. In *Design Automation Conference*, 2002.

[47] Xipeng Xiao and Lionel M. Ni. Internet QoS: A big picture. *IEEE Network*, 13:8–18, 1999.

[48] Wenjiang Zhou, Chuang Lin, Yin Li, and Zhangxi Tan. Queue management for qos provision build on network processor. In *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, page 219, 2003.