Design and performance evaluation of a Programmable Packet Processing Engine (PPE) suitable for high-speed network processors units

K. Vlachos a,*, T. Orphanoudakis b, Y. Papaefthathiou c, N. Nikolaou d, D. Pnevmatikatos c, G. Konstantoulakis e, J.A. Sanchez-P. f

a Computer Engineering and Informatics Department, University of Patras, GR-26500, Rio, Greece
b Telecommunication Science and Technology Department, University of Peloponnese, Greece
c Electronic and Computer Engineering Department, Technical University of Crete, 73100 Chania, Greece
d Ellemedia Technologies, GR 17121, Athens, Greece
e InAccess Networks, GR 17672, Athens, Greece
f Greek Research and Technology Network (GRNET) GR-11527, Athens, Greece

Available online 13 October 2006

Abstract

In this paper, we present a Programmable Packet Processing Engine suitable for deep header processing in high-speed networking systems. The engine, which has been – fabricated as part of a complete network processor, consists of a typical RISC-CPU, whose register file has been modified in order to support efficient context switching, and two simple special-purpose processing units. The engine can be used in a number of network processing units (NPUs), as an alternative to the typical design practice of employing a large number of simple general purpose processors, or in any other embedded system designed to process mainly network protocols. To assess the performance of the engine, we have profiled typical networking applications and a series of experiments were carried out. Further, we have compared the performance of our processing engine to that of two widely used NPUs and show that our proposed packet-processing engine can run specific applications up to three times faster. Moreover, the engine is simpler to be fabricated, less complex in terms of hardware complexity, while it can still be very easily programmed.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Embedded networking systems; Network processor; Special-purpose processor

1. Introduction

The explosive growth of the Internet has created an insatiable demand for bandwidth. The emergence of Wavelength Division Multiplexing (WDM) has increased the backbone capacity to terabits per second, shifting the bottleneck back to the network processing systems, namely routers and relevant switching equipment. Furthermore, the convergence of voice and data networks, as well as the introduction of new Quality-of-Service (QoS) mechanisms and recently developed protocols, require network flexibility that the currently employed hardware units cannot probably provide.

A promising solution to both problems was provided by the introduction of a new class of Embedded Integrated Circuits called Network Processing Units or NPUs. NPUs are becoming the silicon core of every network system that requires a high degree of flexibility to support evolving network services, at extremely high packet rates [1–3]. Whereas legacy architectures for building networking equipment were based either on general-purpose processors (GPPs), which offer high flexibility due to software programmability...
In this paper, we present a novel, Programmable Packet Processing Engine, suitable for deep packet processing in high-speed networking systems. Its architecture mainly falls into the second category described above but its pipelined architecture has been designed so as to achieve its maximum performance irrespective of protocol state dependencies and minimize context switching overheads. The engine can be used in most available networking units and systems, so as to replace a number of general purpose CPUs used. The proposed design is a flexible and programmable engine that can sustain multi-Gigabit wire speed protocol processing, even for complex and highly demanding networking tasks. This engine has been developed as the basis of the PRO3 network processor [10] and a top level design appears in [11]. In this paper, we present a detailed block description of the sophisticated packet processing engine (PPE) and its sub-blocks, and we also present the individual as well as its overall performance for a number of applications. The objective of this paper is to illustrate the efficiency of our Programmable Processing Engine even outside the fabricated PRO3 device as a generic Processing Engine for Network Processing Units. Additionally, in this paper we present a detailed profile of various networking applications providing detailed insight on the issue of application programming based on this processing architecture.

In particular, the contributions of this paper can be summarized in: (1) the architecture of this sophisticated network processing sub-system, which proved to be very efficient when executing a number of applications, (2) profiling of a number of real-life network applications, showing which parts of them are CPU intensive and which of them can be significantly accelerated using specific purpose processing units, instead of general-purpose ones, and (3) the micro-architecture of the various sub-modules of this engine as well as their most innovative features. We believe that the architecture of the presented processing core, which is shown to have certain advantages compared to a number of similar architectures, can serve as a reference for designing network processing units. Moreover, the comprehensive analysis of a number of real world network applications can serve as a guide to both software developers to build efficient applications and to system designers to implement optimal systems on a cost versus performance basis.

The rest of the paper is organized as follows. Section 2 presents the architecture of the Programmable Packet Processing Engine, hereinafter called PPE, while Section 3 presents its detailed micro-architecture. Section 4 presents performance evaluation results and finally Section 5 compares the proposed design and performance to that of the Intel IXP and HiFn’s Power NP.

2. The Programmable Packet Processing Engine – functional architecture

The PPE module has been developed as the main processing engine of the PRO3 network processor, with a target capacity of 2.5 Gb/s (OC-48). The PRO3 architecture,
shown in Fig. 1, aims at accelerating the execution of transport or higher layer protocols by extending a high-performance RISC-CPU with programmable, pipelined hardware. PRO3 combines two levels of functionality: (i) it processes at wire speed low layer protocols (cell/packet reception, IP and ATM header processing, classification and framing) by using specialized hardware blocks and (ii) it accelerates execution of higher layer protocols by integrating specially enhanced, custom made processing cores for field processing. It mainly consists of high-performance application-specific hardware modules, as well as special-purpose and general-purpose processors for the data plane and the control plane, respectively. CPU demanding and (hard) real-time protocol functions are handled by the special purpose processors and the application specific hardware, while the remaining functions, as well as higher layer protocols are handled by the on-chip general purpose CPUs in an integrated way.

The Packet Processing Engine (PPE) is the processing heart of the PRO3 system and handles the execution either of entire network protocols or of the most frequently-used (i.e. assuming error-free operation, which is the normal case for an operational network) and time-consuming functions. This is done according to the requirements of each application, as well as the type and protocol of the packet, as discussed in the following subsections. PPE is a three-stage pipeline module consisting of three logical sub-units: a header Field Extraction Engine (FEX), a modified RISC-CPU core and a header Field Modification Engine (FMO).

In addition an I/O Data Controller is used to relieve the processing core from I/O duties and free available resources for pure processing tasks. Fig. 2 displays the functional model of the PPE unit.

The modified RISC-CPU (MHY) is a derivative of the standard RISC Hyperstone E1-32XS microprocessor core [12]. It uses 32 global and 64 local registers of 32 bits each, 16 global and 16 local registers directly addressable. Two sets of 14 global registers and 64 local registers are accessible from the PPE controller via a special port. The register file of this CPU is divided into two parts, and while the core processes data on one part, the Data Controller (specialized hardware) writes to or reads from the other part of the register file, new fields that are coming from or going to the FEX or FMO, respectively. In this way, data processing and data I/O operation are executed in parallel and therefore, the MHY is not stalled at any time, whereas in other networking systems, the processing units perform the data movements themselves. Similarly, the MHY cache has an external port, and therefore the state of the network flows is written to it without the need of any intervention from the MHY. The MHY processes the data directly from its cache and writes the updated state back to it. The CPU-core accesses are switchable between the two sets of 14 global registers and between the two halves of the 64 local
registers. By parallelizing, the I/O operations with the packet processing, the latencies imposed by I/O operations are completely hidden.

This PPE design yields significant advantages in a large number of processing tasks. In summary, the use and efficiency of a typical processor core is enhanced by providing the hardware means to tailor its circuits for specific tasks. Additionally the diversity of applications that can efficiently be executed in the highly optimized header processing (fields extraction and modification) units is greatly broadened.

An important component of the PPE architecture is the Packet Delay FIFO. The received packet or part of a packet, after the field extraction, is temporarily stored in the Packet Delay FIFO waiting for the processing results. After, FMO receives the results (new fields and commands) from the Data Controller -depending on the application and the processing result – it may (a) reject (b) modify or (c) construct a new packet (or message) and store it or send it to another unit or even again to PPE for further processing.

The PPE unit is designed so as to perform field processing using fully programmable engines that operate on a protocol-based firmware, which is stored in an on-chip memory. Especially the instruction set of the custom made FEX and FMO comprises of simple and generic, protocol-independent instructions that can be applied in any protocol or protocol encapsulation format [11]. In the PRO3 there are two identical PPEs and data processing is balanced between them.

Before we proceed to describe the PPE micro-architecture in the following section, a brief note should be made here regarding the assumptions that were taken into account related to the overall memory architecture of the NPU where the PPE is integrated. Although the memory organization and access scheme is an integral part of an NPU where the PPE is integrated. Although the memory organization and access scheme is an integral part of the hardware architecture of the main processing element, i.e. the PPE.

For reasons of coherency and completeness we summarize here the main features of the overall PRO3 NPU memory organization, which is partitioned in three main memory blocks. The first is related to the flow classification and look-up table searching issue, the second to the per-flow packet queuing requirements and the third to the per-flow, protocol and application state maintenance. Flow classification must have been completed before any processing can take place and frequently requires longest prefix matching on one or several fields. Since this is a difficult and custom task the majority of the current NPUs employ a dedicated packet classifier component based either on custom hardware or interfaces to appropriate peripherals/coprocessors. The PRO3 NPU implemented this function externally to the PPE employing a dedicated FEX engine integrated to a Ternary Content Addressable Memory (TCAM) controller. Packets are classified upon reception into flows, stored in per-flow-queues in the external packet buffer memory and are communicated to the PPEs (most usually only the packet header is required) by the Data Memory Management (DMM) unit (initially upon the arrival of new packets from the network interfaces this is initiated through the intervention and control of an internal task-scheduling unit). Packets can be flexibly accessed by means of the DMM, which communicates with the processing elements by means of commands and data buffer transfers (up to 64-byte bursts) through the internal system bus. Finally state is maintained per flow in an external memory (control RAM in Fig. 1), which is organized in pages of programmable length and indexed by the flow identifier. The PPEs operate on the information retrieved from and stored in these three external locations. A fourth memory block and the respective dedicated unit has to do with packet scheduling and custom traffic management functions, which is off-loaded in this way from the application software.

3. PPE physical decomposition

In this section, PPE is physically decomposed and its main building blocks are presented in detail. The overall micro-architecture of the PPE block is shown in Fig. 3. The main characteristic of the entire PPE is that processing time consists of Processing Slots. Each Processing Slot varies with time and is able to accommodate and process a packet or part of a packet, up to seven 64-byte segments.

3.1. I/O Data Controller

As mentioned previously, PPE uses an intelligent I/O subsystem to offload the transfer of packet data and state information from the processor core. Fig. 4 displays the functional architecture of the I/O Data Controller. The control logic receives the extracted fields from the FEX engine, matches the incoming packet with the flow state, reads the required fields from memory and passes it directly to the processor registers. As already mentioned in the previous section, the processor register file is divided in two parts and while the RISC core processes the data at one part, the Data Controller writes or reads new fields to/from the other part. In this way the RISC (MHY) performance is significantly enhanced as I/O operations are performed in parallel with the packet processing of the previous slot. An important feature of using such an I/O Data Controller is that it facilitates the development of software since existing tool-chains (compilers, etc), can be easily augmented to utilize the added features.

When all the necessary information is available, the processor commences protocol processing using its registers. The results are also stored in registers, and the control logic extracts the data and transmits it, if necessary, to the FMO engine. A feedback signal from the MHY to the entire PPE can extend the processing cycle of a certain packet. This may stall the operation of the PPE in order to accommodate extended packet processing requirements.
For successful operation, the hardware and software coordinate in two ways:

- When a packet arrives, the hardware must notify the software to initiate packet processing; and
- When the processing is completed, the software must notify the hardware to send the processed packet to the output.

To address the first requirement, a simple handshake scheme is used: A signal from the control logic notifies the processor that a new packet is available for processing; the processor maintains an idle signal, which informs the control logic that it is not processing any packet. Deactivation of the idle signal indicates the beginning of packet processing. Reactivation of the idle signal indicates the completion of processing, and the processed packet can be transmitted to the FMO module.

On the output side, this controller is more involved. To output a packet, the control logic must know its size, where to send it, and some related information. Moreover, the result of processing can be more than one packet. In order to handle these cases, we have defined a format for a software result register. This is the first register that the output logic always reads, and it defines all subsequent necessary actions. The register is logically divided into the fields listed in Table 1. For multiple output packets, the interface uses one software result register per packet.

The proposed hardware-software interface is register-based and easily expressible in software with a function (or maybe a system) call that on return updates the appropriate registers. Therefore, even with the defined interface,
designers can use traditional compiler tools to develop, optimize, and debug the application code. This advantage can be crucial for timely product development.

The I/O Data Controller micro-architecture, as shown in Fig. 3b, includes two units: the read-write control RAM (RWR), and the modified RISC Processor Glue logic (RPG). The input module transfers packets and flow state into MHY’s register file for processing. The output module transfers the process results (fields, flow state, and commands) to the field modifier module. The input module also supports internal-state bypass, detecting the location of the latest version of the flow state for each incoming packet. Because processing is pipelined, the correct state information is available in either the MHY’s register file (just updated from the previous packet), or in the FIFO RWR buffers (the state was updated but is not yet written to memory), or in the FIFO state buffer. The RWR module performs three major tasks. For each packet, it

- Reads the appropriate state information from the control RAM and transfers it to the RPG’s input sub-module,
- Receives the updated state from the newly processed packet and writes it to the control RAM, and
- Acts as a searchable write buffer to ensure that reading the control RAM always provides the most up-to-date results.

### 3.2. The field extraction and field modification Engines

The FEX and FMO engines are small custom RISC cores, with an internal three-stage pipeline architecture. Their micro-architecture is shown in Fig. 3a and c. Although not identical, their logical design is quite similar. They are fully programmable and operate on a protocol or application specific firmware. Only specific fields are extracted/modified from the received (whole or part) packet. The payload is very rarely sent to the PPE for processing since the majority of the networking applications operate on the packets’ headers only.

The FEX engine alleviates the MHY from packet verification, bit and byte processing, and leave only generic software execution to it. As shown in the performance section, this results in an almost constant ratio of cycle budget to packet length for the majority of the applications and reduced total processing time. A field extraction operation may start by accessing the first (header) or last 64-byte segment of the packet fetched from the packet buffer (a transfer scheduled during packet reception). Alternatively, operation may be initiated by MHY software in the case of deep packet processing and in this case various segments of the received packet are sequentially fetched for processing in the FEX-FMO engines. It is worth noting here that this case is the only one that may stall the pipelined operation of PPE (and only for specific protocol state dependences). In all other cases the PPE can process packets from the same or different flows back-to-back.

FEX has been designed as a custom RISC for bit and byte processing. It executes 9 basic instructions and 4 optional commands. The instructions are used to parse the data, while the commands are used to control the internal registers. FEX uses 4 generic registers, a Program Counter and a Data Pointer. The instructions are: NOP; EXTRACT \((n,b)\) that extracts a field of \(n+1\) bits with rightmost being bit \(b\); MOVE A to DP; MOVE B to DP; ADD A to DP; ADD B to DP; JMP C, a, addr; JMP D, a, addr (if C or D is equal to a then jump to addr); STR (restart); and FLGS type, flags (sent flags to the rest of the PPE). The commands are: DEC DP; INC DP; DEC A; and DEC B. The optional commands are performed in parallel with the basic instructions. Each of the basic instructions can be combined with one or more of the optional commands. It should be noted that the instruction and the active commands are executed concurrently. In the current implementation the FEX, executes firmware of up to 2K-instruction that is sufficient enough for the typical network applications. Specific tools for programming FEX have been developed.

The FMO engine is similar to FEX but it performs the dual task. It adopts a three-stage pipeline architecture as well. FMO aim is to compose the protocol message (byte stream) according to the protocol specifications (firmware controlled), taking as input the results of processing from the I/O Data Controller (fields) and the data stored in the Delay FIFO. The FMO performs one of the following: (i) reject the packet(s) in case of errors, (ii) modify the original packet with the fields received from the I/O Data Controller, and/or (iii) compose a new packet. The FMO engine executes 16 basic instructions and 6 optional commands. Each instruction can be combined with one or more of the commands. The instructions are used to parse the data from the Delay FIFO and compose the new packet, while the commands are used to increase or decrease the internal pointers/registers. FMO uses 5 registers (A–E), a working register (WR), a Program Counter and a Data Pointer as depicted in Fig. 3c.

The instruction set consists of 16 basic instructions and 6 optional commands. The instructions are: NOP, MODIFY \((n,b)\) that modifies a field of \(n+1\) bits with rightmost being bit \(b\), MOVE E to WR \((n,b)\) that replaces \(n+1\) bits of WR register (left most bit is bit \(b\)) with the respective bits of register A, LOAD E with \((n,b)\) that loads register A high \((b=1)\) or A low \((b=0)\) part with the 16-bit value n, JOE X \((a, addr)\) where X is one of the registers A, B, C or D, that jumps (loads Program Counter) to address \(addr\) in the instruction memory if \(a\) is equal to the value of register X,

<table>
<thead>
<tr>
<th>Application</th>
<th>IPv4 forwarding</th>
<th>DiffServ</th>
<th>IPF/NAT for TCP packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average number of instructions</td>
<td>29</td>
<td>58</td>
<td>113</td>
</tr>
</tbody>
</table>

---

K. Vlachos et al. / Microprocessors and Microsystems 31 (2007) 188–199
JNE X (a, addr) that jumps (loads Program Counter) to address addr in the instruction memory if a is NOT equal to the value of register X. START that causes FMO to start a new program execution (Program Counter is reset) as soon as a new packet (or part of packet) is available, SEND that sends the content of the Data Memory buffer to the output bus and EXTRACT (n, b) that operates on the word read by the previous instruction and extracts a field of n + 1 bits with rightmost being bit b. The optional commands are the following: RD that reads the next word from Delay FIFO to WR, WR that sends the content of WR to Data Memory and DEC X, where X is one of the available registers A, B, C and D that decrements its content by. The instruction and the commands are executed in parallel. The FMO is able to efficiently compose new packets or modify the Delay FIFO packet based on the MHY processing results.

Within the PRO3 NPU, both engines, FEX and FMO, are connected to a bus with a maximum throughput of 6.4 Gbps. This is accomplished by using a 32-bit wide data path at an operating clock frequency of 200 MHz (System Clock of the current implementation). The average throughput may be less, since some instructions need more clock cycles to be executed and further since one 32-bit word may consist of several fields received separately. The average cycle-to-instruction ratio for both engines for typical network applications was measured to be 1.6 for the FMO and 1.4 for the FEX [11]. A clear advantage of this architecture is that the hardware complexity is substantially lower compared to the general-purpose engines used in the majority of the existing – or proposed – NPUs (8.3 K CMOS gates for the FEX, 19.8 K CMOS gates for the FMO and 90.3 K CMOS gates for the combined module of I/O Data Controller and modified RISC-CPU). Furthermore in terms of performance as will be shown in Section 4, the FEX and FMO engines can perform their processing tasks up to five times faster than the 32-bit RISC engines incorporated in commercial widely used NPUs, while they cover a silicon area similar to that of an 8051 compatible, 8-bit micro-controller core as the one in [20].

4. Performance evaluation

In this section, we first discuss the decomposition of three typical networking applications into software for a conventional RISC architecture, and into firmware for the FEX and FMO modules. Then we present performance evaluation results for each sub-module of the PPE engine. In our studies, we have used real TCP/IP traffic as input and the presented results were derived from extensive simulations of the RTL model of the PPEs within the fabricated PRO3 NPU. Experiments were carried out, developing firmware for all the micro-engines, porting open source C code to the modified MHY, where available and developing our own software for the rest of the applications. The main quantity measured was the processing time and the total instruction number executed in all PPE modules namely, the FEX, FMO and the MHY core. Based on these measurements, the actual data throughput of each sub-block as well as of the whole PPE unit was derived, expressed in packets/s – or in bit/s, for given packet sizes. Average numbers are also used, when measuring the instruction count of a PPE module for a certain application, since they depict the overall application level performance, i.e. data forwarding throughput. These average values have been calculated with traffic patterns derived from large backbone traces of various USA network providers [13]. Using these traffic statistics, the average number of instructions executed and the achieved throughput was calculated. In the analysis all the overhead messages were included (setup and inter-communication overheads).

It is worth noting here that the number of executed instructions per unit of time, in terms of Million Instructions Per Second (MIPS), as a performance metric, cannot be used for processors/NPUs belonging to different categories, since their efficiency is directly affected by their instruction set architectures. This is because MIPS as a metric provides the number of executed instructions and not the completed processing tasks. However, MIPS can provide useful insight regarding the relative performance of different CPUs. In order to elaborate further on the issue of NPU benchmarking, in the next subsection the benchmarking efforts within the Network Processing Forum are summarized.

4.1. NPU benchmarking

In order to administrate the benchmarking efforts, the Network Processing Forum (NPF) has emerged, as a result of two pre-existing industrial groups. One of the main concerns of NPF was to create a standard benchmark for accurately measuring the power of these complex ICs, since their actual performance can be measured across different dimensions. Until now, NPF has released benchmarks for best effort packet forwarding applications, including IP routing (recently augmented to include IPv6) and MPLS forwarding. The implementation of the PRO3 NPU has taken into account several potential applications that ranged from processing of Q.2931 over AAL5 over ATM signaling messages in a high-capacity ATM-based Access Concentrator to multi Gbit/s capacity Firewalls, traffic inspection engines network gateways etc. For our evaluation we selected a set of applications that could demonstrate the throughput upper bounds under different traffic loads and provide results comparable to those published in the literature. Within this context we focused on IP-based and data processing applications rather than control plane processing, since in the latter case there is no strict metric related to the so-called “wire-speed” performance (control plane protocols are expected to consume a portion of the link bandwidth depending on the deployment scenario).

Since plain IPv4 routing was the first benchmark used by the majority of the NPU vendors and examined in the literature we have also used it as an example application
scenario. However, in order to explore the performance of the proposed processing unit in applications involving complex per flow processing and state manipulation, we have also implemented two widely used applications. The results shown in the next section expose the enhancements introduced by our architectural approach and implementation. In particular, the three applications we implemented, represent typical examples that involve intensive packet processing and require the increased power of an NPU to sustain the requested throughput, while two of them (IP routing and NAT/Firewall) are also included in the NetBench benchmarking suite for Network Processors [14].

4.2. PPE benchmarking applications

The first application is IPv4 Packet Forwarding, a relatively simple application in terms of processing power. The second application is packet forwarding in a DiffServ domain and involves packet classification, metering and marking as executed in a DiffServ Edge Router. Finally, the third and most demanding application is a stateful inspection firewall with Network Address Translation (NAT).

In the IP forwarding case, PPE software mainly implements a header modification function. In this case, the PPE must decrement the Time To Live (TTL) field of the IP header and accordingly modify the value of the header checksum. In the DiffServ edge router application, apart from the default header checks and the classification decision for packets entering a DiffServ domain, the output of the header modification stage is also controlled by a rate metering function. In particular, the PPE software implements the packet metering and marking functionality of an edge DiffServ capable router as defined in RFC2475 [15] and RFC2697 [16]. For the Firewall application we used the open source implementation of the IP Filter (IPF) as a reference [17]. IPF is an open source packet-filtering engine with Network Address Translation (NAT). Apart from static packet filtering based on Access Control List (ACL) rules, the most complex function of the IPF is the stateful inspection task of the TCP connections. IPF follows the transitions of each endpoint of a TCP connection by monitoring the content of both the IP and TCP headers of all TCP packets, while also keeping track of the protocol state for each session identified. Packets that do not correspond to the current state are dropped. Finally, NAT results in checksum recalculation both of the IP and the TCP headers.

4.3. PPE evaluation results

Fig. 5 shows the results measured for the custom field processing engines, FEX and FMO, of the PPE for the IPv4 forwarding and the DiffServ metering applications. Both applications present similar header processing requirements since they are based only on the IPv4 header information and this results in a reduced number of bit and byte level operations that have to be executed. To this end, only a single data segment, that includes the IPv4 header, must be retrieved from the NPU data buffer, and thus the number of instructions executed in the FEX engine is constant.

On the contrary, FMO executes a higher and variable number of instructions that depends on the IP header length. This is due, on one hand, to the delineation need of the IP header in the I/O controller memory so as to perform header modification and, on the other hand, to the need to construct the appropriate internal system commands to move the processed data/commands to the separate external memories. When the traffic consists of minimum size IP Packets, FMO needs about 30 instructions to manipulate one packet. This takes about 46 clock cycles and thus the maximum bandwidth supported by FMO engine is close to 4.25 Mpps.

Moving to the MHY, it performs two operations in the IPv4 forwarding application: a decrement in the TTL field of the header and a checksum recalculation. Using an incremental procedure to recalculate the checksum, and optimizing the code for speed efficiency (trading-off code size, which is not critical in our case) MHY executes these operations in only 29 clock cycles. Table 1 shows the MHY number of instructions/clock cycles for each application.

To this end, we may conclude that the PPE overall throughput in IPv4 forwarding application is 4.25 Mpps, as

![Diagram](image-url)
defined by the worst case throughput of the FMO engine. Therefore, a single PPE engine can sustain wire-speed processing for up to 1.8 Gb/s, in the worst case, when the input traffic stream consists of only 40-byte minimum-size packets. To project the FMO performance in a complete NPU implementation we refer again to the PRO3 NPU, where two PPEs have been implemented operating in parallel. Thus, the aggregate serviced throughput, even in this worst case exceeds by far the 2.5 Gb/s rate, which was the target capacity. It is also worth noting here that the VLSI area of the PPE was not restrictive in the PRO3 implementation (which was a pad-limited design with 912 I/Os, comprising of 865 K gates in total, while the core was occupying a 53 mm² area) and the architectural choice for just two PPEs was made after judging the overall performance adequate.

In the DiffServ application, MHY is responsible for executing the actual leaky bucket policing [16] and the checksum recalculation after header marking. These operations are performed in 58 clock cycles (see Table 1). Therefore, in this application, the slowest pipeline stage of the PPE engine is the MHY and as a result the performance of the entire PPE is defined by MHY. Therefore, the PPE can process one packet in approximately 60 clock cycles or about 3.35 Mpps (one or two extra clock cycles are needed for data storing and forwarding between the pipeline stages). Again, for the parallel implementation of two PPEs, the aggregated throughput increases to 6.75 Mpps and thus the 2.5 Gb/s input rate can be sustained even in the worst case of a constant traffic stream with minimum size packets.

In the IP Firewall with NAT application, processing in the PPE is more complex. According to the IPF specifications, address translation requires calculations and modifications on both the IP and the higher layer protocol headers (TCP, UDP and ICMP protocol encapsulations are examined). To this either one or even two segments of the buffered packet is fetched for processing, since the TCP header may reside in a second 64-byte segment. Subsequently, FEX extracts the IP-Header-Length and the IP-Packet-Length fields from the IP header, as well as the TCP-Sequence-Number, ACK-Number, Window-Size, Offset, Flags, and Checksum from the TCP header. These 8 fields are then transferred to the MHY via the I/O controller for processing. Fig. 6a shows the total number of instructions executed by FEX for a single (64-byte) or double (up to 128-byte) segment. The number of instructions is proportional to the IP header length, which in turn depends on the number of valid IP options. The total number of instructions for two segments (when the IP together with TCP header length is larger than 64-bytes) is smaller than that of one segment since the firmware easily identifies the case of two segments and jumps directly to the fields to be extracted and which reside in the second segment. Fig. 6b shows the equivalent FEX throughput in each case. From both figures, we may conclude that a single FEX engine is capable of processing packets at half the 2.5 Gb/s rate and hence the aggregated throughput with parallel operation of the two PPEs can reach the targeted OC-48 rate. The FMO receives updated header fields from the MHY via the I/O Data Controller and composes new packets for transmission. Fig. 6c shows the number of executed instructions versus the total IP packet length for different IP header lengths. It can be seen that the number of FMO instructions executed is proportional to the IP packet length for packets with the same IP header length. Additionally, for IP packets of the same length, the number of instructions is inversely proportional to the IP header length. This is because when many IP options are valid, fewer jump instructions are needed to scan the contents of the packet.

![Fig. 6. (a,c) Number of instructions, executed in Field Extraction and Field Modification engine. (b,d) Corresponding throughput for IPF application.](image-url)
Finally, Fig. 6d shows the equivalent FMO throughput for each case.

In the IP Firewall with NAT support application, the TCP processing (which includes the stateful inspection task) running on the MHY is the most complex part of the executed code. The main body of the TCP state processing, the “inner loop” of the application, can be logically separated into two sections: the first consists of a number of calculations (addition, subtraction and multiplication) and comparisons, called the “Calculate and Compare” section and the second, called the “Switch” section, is a large switch statement. After optimisation, this function yielded 110 instructions for the “Calculate and Compare” and 60 instructions for the “Switch” section. It should be noted that neither all instructions of the “Switch” section nor of the “Calculate and Compare” section are executed for each packet. On average, as shown in Table 1, 113 clock cycles are needed for the TCP stateful inspection with NAT. Operating at 200 MHz, this results to an average execution time of 565 ns per packet or almost 2 Mpps.

4.4. Workload against data throughput

As it has been mentioned above the performance of the software depends on the size of the packet headers as well as on the protocol type. To this end, small packets represent a heavier workload for the PPE and since the application level throughput is the actual metric perceived by the system user, we translated the above profiling results to actual data throughput for real packet streams found in the Internet backbone today. To this end, we developed a traffic generator using samples of the aforementioned traffic patterns,[13,18], and measured the MHY performance. Fig. 7 shows the throughput supported by a single MHY for different packet sizes, when executing the most complex of the aforementioned applications (i.e. the IPF/NAT). It can be seen that two PPEs can sustain processing at a 2.5 Gb/s rate, if the network streams comprises of packets with an average length of 128 bytes (which is less than the typical mean IP packet length of 273 bytes [13]).

Fig. 8 shows the throughput achieved for the average and worst case. In particular curve (a) shows the case, when the percentage of the small packets (less than 100 bytes) in the pattern increases. The traffic generator is set to produce small and large packets based on a Gaussian distribution with a mean value of 70 and 170, respectively. From curve (a), it is clear that even when the flow consists of 90% of small packets the software can process the data at the requested speed. We have also investigated the performance of MHY when each of the packets (irrespective of their sizes) needs the worst case number of instructions, that is 167 instructions, to be processed (i.e. worst case processing is valid only for a very small set of real-world packets). Curve (b) of Fig. 8 shows the corresponding throughput achieved. It can be seen that PPE is capable of processing at the peak rate, only if input traffic comprises of at least of 50% of packets with length greater than 100 bytes. It is worth noticing here that for obtaining Fig. 8 results, packet sizes close to the statistics of real networks (as in [18] and [13]) were produced. As a result, even though all packets request the worst case processing-time,
PPE is capable of sustaining processing at the peak rate as long as packet length distribution is similar to that of a typical IP network.

Another interesting aspect of the MHY performance is the way throughput varies with the network protocol type. Fig. 9 demonstrates the bandwidth achieved when a network stream of TCP and UDP packets is processed. From Fig. 9, it is clear that two PPE modules can sustain processing at the peak rate, when more than 20% of the traffic comprises of UDP packets. Again this is consistent with network statistics reporting that at least 20% of the traffic is UDP packets and thus even when packets are short (mean length of 69 bytes) two PPEs can meet the requirements for a 2.5 Gb/s throughput.

Overall, the efficiency of the Field Extraction/Modification engines and their innovative interfaces is clearly demonstrated. Even in the IPF/NAT application, the MHY core is the performance bottleneck, albeit it is relieved from header prep- and post-processing tasks. In principle, incorporating a faster RISC-CPU and following the same architectural concept a much higher throughput can be achieved.

5. Performance comparison with state of the art Network Processor

In order to compare the PPE performance with that of the Intel IXP1200’s processing cores, we have used the IXP1200’s performance results from [19] and executed the aforementioned applications on an IXP platform. In [19], the performance of the IXP1200 is analyzed, when executing simple vanilla IP Forwarding. Since the only performance evaluation metric given is MIPS, we express the above performance results of PPE in equivalent IXP1200 MIPS. According to [19], even if the new MAC address field is available (no memory overhead) the IXP1200 can sustain up to 3.41 Mpps processing, using all its processing capability that is quoted as 1200MIPS. As a result, and according to the results obtained in Section 4, a single PPE with a processing capability of 4.55 Mpps offers about 150% the processing power of an IXP 1200, or 1600 IXP equivalent MIPS.

In the case of the IXP-2400 that is manufactured on the next generation CMOS 0.13 µm process – against the 0.18 µm used for PPE – its power is reported to be 4800 IXP MIPS using all its eight, 600 MHz processing units. In other words two 200 MHz PPEs offer 75% of the processing power of the full IXP (while being significantly less complex in terms of hardware implementation). Regarding DiffServ application, IXP-2400 performance is found to be equivalent to two 200 MHz PPEs [21].

In order to be able to measure the performance of the Intel IXP2400 when executing the IP/NAT application we have ported our code to this network processor, using the Intel development platform and tools. We opted for employing three of its micro-engines, executing the equivalent of FEX, FMO and MHY programs, respectively, connected in a 3-stage pipeline. This was found to be the best, performance-wise configuration for IXP as well, since it balances the workload almost equally to all three engines. For the UDP case there is no need for address translation and recalculations, and the optimal configuration proved to be 2 IXP micro-engines forming a two-stage pipeline. Table 2 shows the instruction count numbers for the IXP2400 obtained, applying the highest compiler optimizations for performance and handcrafting the produced code, so as to further increase the execution speed. The throughput numbers quoted include also the inter-communication and scheduling overheads when executing these applications. According to Table 2, two PPEs can support in total up to 3540 Kpps in the TCP/NAT case, while the whole IXP system up to 4571 Kpps. Similarly, in the UDP case the two PPEs can handle 4300 Kpps, while the whole IXP-2400 up to 9567 Mpps. As it is clearly shown, the optimization merely stems from the efficiency of the FEX and FMO instruction sets for certain protocol processing tasks, since for the IXP’s micro-engines the same functions need about 3 times more instructions and clock cycles. Additionally, the

---

Table 2: Mean instruction count and mean clock cycles for RPMs and IXP2400

<table>
<thead>
<tr>
<th></th>
<th>TCP NAT</th>
<th>UDP no NAT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instructions</td>
<td>Clock cycles</td>
</tr>
<tr>
<td>PRO3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FEX</td>
<td>58</td>
<td>94</td>
</tr>
<tr>
<td>FMO</td>
<td>70</td>
<td>112</td>
</tr>
<tr>
<td>MHY</td>
<td>108</td>
<td>113</td>
</tr>
<tr>
<td>RPM (slowest stage)</td>
<td>113</td>
<td>1770</td>
</tr>
<tr>
<td>PRO3 (2xRPMs)</td>
<td>3540</td>
<td></td>
</tr>
<tr>
<td>IXP2400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 µengine (FEX program)</td>
<td>227</td>
<td>297</td>
</tr>
<tr>
<td>1 µengine (FMO program)</td>
<td>272</td>
<td>326</td>
</tr>
<tr>
<td>1 µengine (PPE program)</td>
<td>170</td>
<td>230</td>
</tr>
<tr>
<td>µengines complex (slowest µengine)</td>
<td>326 (3 µengines)</td>
<td>1840 (3 µengines)</td>
</tr>
<tr>
<td>IXP (8 µengines)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 Note that when measuring the aggregate IXP performance with 8 micro-engines, one of them executes both the FEX and PPE code while the other seven are running a single program.
low instruction count and latency of the complete PPE expose the efficiency of the innovative direct register access I/O scheme. Table 2: Mean.

Looking at the formerly IBM’s and now HiFn’s Power NP4 family, it uses dedicated hardware modules consisting of general-purpose processors that are optimized for a specific task, while the protocol processing is performed by simple general purpose processors (called PicoProcessors). The PowerNP can support up to 2.5 Gb/s TCP/IP traffic consisting of minimum size packets when executing vanilla IP Forwarding and it consumes close to 80% of its total processing power [9]. Since the quoted general purpose processing power of the PowerNP4GS3 is 2128 PowerNP MIPS, the vanilla IP Forwarding in this NPU consumes about 1700 MIPS at a rate of 6.75 Mpps. Since a single PPE executes IP Vanilla Forwarding at a peak rate of 4.55 Mpps, its processing capacity can be compared to the equivalent of 1700 x 4.55/6.75 or about 1100 PowerNP MIPS. In other words two PPEs have the equivalent processing power of about 2200 PowerNP MIPS and which corresponds to the processing power of about 16 133 MHz Picoprocesors.

6. Conclusions

In this paper, we have presented the design and the micro-architecture of a programmable, packet processing engine, called the PPE. The PPE is suitable for high-speed networking systems and uses an innovative concept of a 3-stage pipelined module, integrating a RISC-CPU core with special purpose programmable hardware on the same platform. In particular, the proposed engine consists of: a header field extractor (FEX), a header field modifier (FMO) and a modified RISC-CPU (MHY), combined with an I/O Data Controller, commissioned to offload the latter from input/output operations. Novel aspects of the PPE engine is the register file split of the RISC-CPU in two separate parts, so as to allow the core to process data on one part while the I/O Data Controller writes or reads fields to/from the other part of the register file. The PPE performance has been assessed mapping three frequently used network applications namely IPv4 forwarding, DiffServ Metering/Marking and IP stateful inspection Firewall with Network Address Translation. The results obtained verify that PPE is highly efficient in such applications and thus can be used as the processing heart in a variety of NP units. Additionally PPE performance has been compared to that of the processing engines of other NPU’s and was found that it provides a much higher performance-to-hardware complexity ratio.

The PPE has been fabricated within a highly sophisticated network processor at UMC’s 0.18 m CMOS process, operating at 200 MHz and occupying about 9.2 mm of area of which about 6.5 mm are covered by the modified RISC-CPU core and its associated memories and the rest by the custom designed special purpose hardware blocks.

References