Scheduling components for multi-gigabit network SoCs

Theofanis Orphanoudakis[a], George Kornaros[a], Ioannis Papaefstathiou[b], Helen-Catherine Leligou[c], Stelios Perissakis[a], Nick Zervos[a]
a) Ellemedia Technologies 223 Siggrou Av, GR17121, Athens, Greece
b) Foundation of Research & Technology Hellas (FORTH), Institute of Computer Science (ICS), Vassilika Vouton, GR71110, Iraklio, Crete, GREECE
c) National Technical University of Athens Telecom Lab, Polytechniopolis Zographou GR15753, Athens, Greece

ABSTRACT

To meet the demand for higher performance, flexibility, and economy in today's state-of-the-art networks, great emphasis is placed on unconventional hardware architectures of network processors. This paper analyzes the problem of processor internal resource and traffic management and proposes a programmable scheduler architecture implemented in a novel protocol processor (PRO3) that deals with the above problems in an integrated way. We briefly outline the architecture of the protocol processor and we support that the innovative scheduling scheme integrated in PRO3 is, in general, crucial for network Systems-on-Chip (SoCs) since it makes it feasible to use external memories for scheduling and still accommodate multi-gigabit network speeds. Extensions to the PRO3 scheduler's architecture are discussed that lead to efficient integration of the component to different network processor architectures at a similar cost. Its beneficial features are easy hardware implementation, low memory bandwidth requirements and high flexibility so as to support multiple service disciplines in a programmable way, thousands of flows and even perform different scheduling tasks.

Keywords: Network processor, scheduling, programmable hardware, SoC

1 INTRODUCTION

Nowadays network processors must employ special schemes in order to handle wire speed cell packet processing at 2.5Gbps and higher line rate, even with state-of-the-art semiconductor technology and advanced circuit design techniques. In addition, besides best-effort applications, todays and future applications require from the network servers to guarantee them some particular performance bound and delay guarantees. The employment of multiple processing elements to accelerate protocol processing in programmable hardware components complicates the resource management problem encountered in modern networking nodes that target efficient resource utilization and Quality-of-Service (QoS) guarantees for service differentiation. The service level that an application enjoys principally depends on how efficiently a network component performs packet/protocol processing and allocates link bandwidth to connections (merely in terms of latency and fairness). To this end, we have developed integrated scheduling components inside a single-chip embedded system: PRO3 (Protocol Programmable Processor). They achieve to facilitate packet processing in a fair, balanced manner and to control the traffic streams generated by the chip, while at the same time are amenable to easy hardware implementation. In the next section we present briefly the PRO3 network processor organization and in section 0 we analyze the scheduling components in coordination to the system's cooperating supporting blocks. In section 4 we describe how our architecture can adapt and exploit its features for achieving higher degree of integration and performance in generic network processor architectures. Finally section0 concludes the paper.

2 THE PRO3 ARCHITECTURE

The PRO3 is designed to achieve high speed processing in demanding networking applications that potentially involve processing of complete protocol stacks (like deep packet processing, stateful inspection, control plane protocol execution etc.). PRO3 combines two levels of functionality: (i) it processes at wire speed lower layer protocols (cell/packet reception, IP and ATM header processing etc.) and (ii) it accelerates execution of higher layer protocols by integrating specially enhanced custom made processing cores for bit and byte processing. Thus the PRO3 system offers three end-to-end processing paths: (i) hardware-based packet reception, storage and forwarding; (ii) wire-speed packet processing involving specialised CPU cores, and (iii) best effort processing involving typical CPUs. Computationally demanding and real-time protocol functions are handled by the programmable hardware, while the remaining functions, including higher layer protocols, are handled by the on-chip or the off-chip CPUs. The concept of the PRO3 architecture is to provide the required processing power through a novel design, incorporating parallelism and pipelining and by integrating generic micro-programmed cores with components optimised for specific protocol processing tasks. Furthermore, efficient scheduling components are integrated so as to facilitate packet processing on a fair, balanced manner as well as to control data streams generated by the chip. The chip can be set up, at boot time, either in ATM mode, where it processes ATM traffic, or in IP mode, operating on variable length packets.

Figure 1 demonstrates the architecture of PRO3. The primary chip I/O's are a pair of 32-bit wide interfaces that can operate either as POS/PHY Level 3, or as Utopia-II interfaces. Fixed-logic packet pre- and post-processors perform lower level functions on ATM cells or variable length packets. Wire-speed packet classification based on multiple protocol fields by means of a RISC-like micro-engine designed for header parsing and programmable Field Extraction and a controller of a high throughput external Ternary CAM (Content Addressable Memory) device for flexible and deterministic classification. The main processing element of PRO3 is the Reprogrammable Pipeline Module (RPM), an innovative module optimised to perform packet processing. This module consists of a bit-field parser (Field Extractor), a CPU with parallelized I/O and processing operations, and a packet constructor (Field Modifier). All together form a powerful 3-stage pipeline module capable of providing the mixed hardware and software processing heart of the system and performing the FSM of each protocol. The implementation of two such modules was selected due to the symmetry of the solution, since they can be independently programmed to execute either the receiver or the transmitter flow of the applications' protocol stacks. The two modules can also execute the same code in parallel facilitating load balancing and increased throughput utilizing the PRO3 shared memory architecture. In this mode of operation the implementation of more than two RPMs is also an option determined by the silicon area and vs. performance trade-offs. Since there is a large number of processing units in the PRO3, after a packet/cell has entered the PRO3 it should be decided in which of them it should be re-directed. This scheduling decision takes into account the fact that some of the units may be busy, and that a particular packet maybe processed faster in one unit than in another one. Especially for this load balancing function we have chosen not to allow data from the same incoming flow to be processed at both RPMs at the same time. This scheduling decision was made so as not to allow one flow to get all the available resources of the PRO3, and to avoid the potential out-of-order arrival of packet/cells at the outgoing interfaces. Obviously this limits the actual bandwidth per flow that can be serviced by the PPE, but we claim that is not a problem since we can still service single flows with rates up to 1.25Gb/sec [1].

The RPMs cannot access directly neither the classification RAM, nor the Traffic Description memories. Their access should be done through the general bus and thus they have to compete with the accesses of the other PRO3 blocks. A scheduling scheme should therefore be involved in prioritizing the accesses. Since as [1] demonstrates the processing units are the bottleneck of this architecture, the RPM accesses should get the highest priority. It should also be noted that an RPM can ask for data from the different memories at any time during the actual protocol processing; whenever such a request from an RPM cannot be satisfied, the pipeline of the RPM should be stalled causing undesired effects.
In order to cope with the high bandwidth memory requirements of the network processing at multi gigabit rates, a specific Memory Management Unit has also been designed (the DMM in Figure 1). The DMM supports a rich set of commands for queue-handling, and can provide up to 10 Gb/sec of sustained bandwidth out of typical memory chips [2]. The integrated scheduling core that is discussed throughout this paper is responsible to regulate packet access operations by means of DMM commands in order to forward traffic to their internal destinations for processing or to the output accordingly, taking into account appropriate QoS parameters. Implementation of timers and protocol context buffering are also a potential bottleneck in generic architectures, therefore in PRO3 they have been offloaded

to dedicated hardware units.

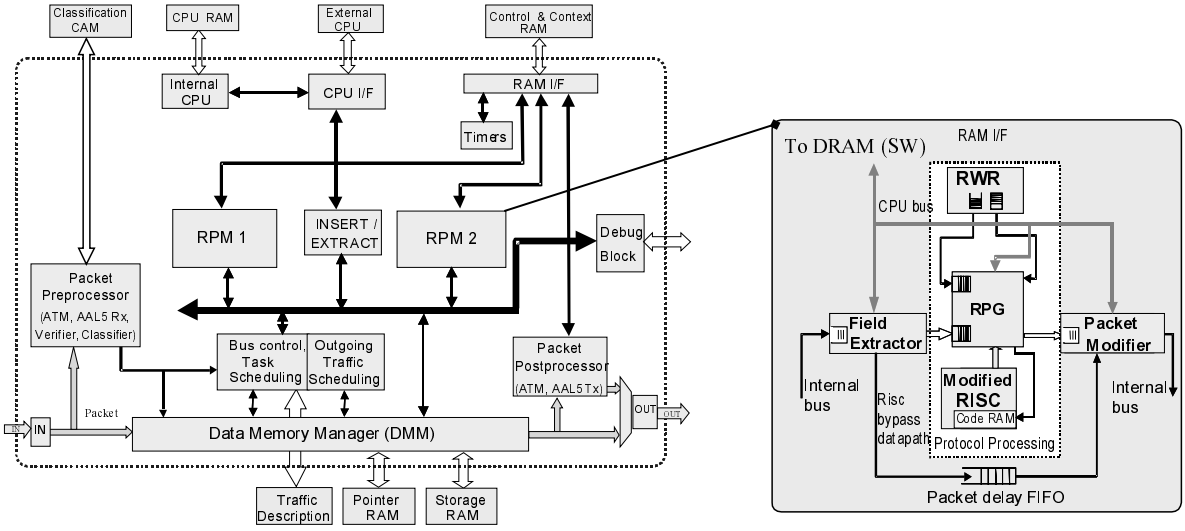

Figure 1: PRO3 Architecture

2.1 Reprogrammable Pipeline Module

The RPM modules constitute one of the main innovations of the PRO3 architecture. Since their architecture imposes operational restrictions on the internal scheduling functionality we discuss their operation in more detail. These modules are used for protocol acceleration and fast path operations, offloading a major part of the main processor's workload, achieving higher performance than a plain parallel implementation of RISC processors would achieve. Each RPM consists of a modified RISC core surrounded by a Field Extraction engine (FEX), which directly loads the required protocol data to the RISC for processing, and a Field Modification engine (FMO) for packet construction and header modification (Figure 1). Together with appropriate interface and control logic, they form a powerful 3-stage pipeline module. The RPM RISC was enhanced so as to optimise the data I/O operation and perform it in parallel with the packet processing of the previous slot. In parallel to the RISC, a FIFO is used to temporarily store received data, waiting for packet processing results. The FMO combines the delayed data with the processing results from the RISC to compose a new packet. This powerful three-stage pipeline parallelizes data I/O with processing and forms the processing core of the system. Obviously the coordination of processing tasks requires specific handling both in terms of software implementation of protocol processing functions on the RPMs as well as by the system control and scheduling functionality in order to achieve coherent protocol processing in an efficient manner.

The modified RISC is a derivative of the standard Hyperstone E1-32XS 32-bit microprocessor core [3]. It uses 32 global and 64 local registers, 16 global and 16 local registers directly addressable. Two sets of 14 global registers and 64 local registers are accessible from the RPM controller via a special port. Core accesses are switchable between the two sets of 14 global registers and between the two halves of the 64 local registers. In this way, the RPM control logic can place state and packet information and retrieve updated data directly into/out of the register file. Placement of protocol fields (extracted from the data packet) as well as protocol state retrieved from the control RAM to the appropriate RISC register set is coordinated by external logic. The control RAM where protocol state resides is accessed per packet by means of the external logic and is organized in the form of pages indexed by the unique internal flow identifier (FlowID) following each packet. The FEX and FMO engines are small custom RISC cores, with a three-stage pipeline architecture. Although not identical, their design is quite similar. They operate on protocol-specific firmware, stored in internal SRAM. The instruction set of FEX consists of 9 basic instructions designed for efficient data parsing, that can be combined with one of 4 optional commands, executed in parallel with the instruction. In the same way, FMO can execute any combination of 16 instructions and 6 optional commands.In

typical operation, FEX receives incoming packets (or packet headers) from the internal PRO3 bus, extracts the desired fields from the header and sends them to the RISC through the special register file port. In this way, the RISC is freed from packet verification, and bit/byte processing, having more cycles available for generic software execution. When processing of the current packet is completed, the RISC switches between the two register file banks and the results of the current packet are read out by the FMO, in parallel with the processing of the next packet. FMO then outputs modified packets to the internal bus. Packet modification is the most complex scenario since specific fields of the stored data in the Delay FIFO may be used to correctly identify the location of the fields to be modified.

2.2 Data Memory Manager

In the reception flow, the operation of the PRO^3 system is based on packet buffering by means of per flow queuing performed by DMM prior to protocol processing. Packets are stored and scheduled for processing by the scheduling module. Each flow is served by the DMM through a dedicated directly indexed by the FlowID value, assigned by the Classifier. The FlowID uniquely identifies the protocol context for each connection and each layer of the protocol stack. The DMM is responsible to buffer incoming data packets, until they are fully processed and ready to be forwarded (or discarded). Variable-length incoming packets are segmented into segments of 64 bytes. The major bottleneck in the data memory manager is the memory utilization. To increase the performance, not only the memory bandwidth, but also the accesses to memory must be highly optimized. To this end, we selected a memory architecture combining large off-chip DRAMs to store segment data, with fast off-chip SRAMs to store data structures implementing queues of segments. We also traded off bank conflicts in DRAMs against memory accesses in SRAMs during real-time storage and forwarding of packets. More details can be found in [2]. The queues managed by the DMM are associated explicitly with one destination within PRO3 and a specific handler/protocol that will be used for the processing of each packet classified in that queue.

To support parallelism in the reception, processing and transmission of packets, the DMM interfaces to the rest of the PRO3 chip through 4 ports: 2 write ports from the input module and the internal bus and 2 read ports towards the internal bus and the output module. Accesses from all ports are served in a round robin manner. Data accesses take the form of commands to the DMM, that may request to enqueue, read, dequeue, or delete one or more packets to/from a specified flow. The DMM interfaces to external DDR DRAM, with sufficient bandwidth to satisfy the multiplexed access streams from the four ports. For 2.5 Gbps line rate the external interface should have a sustained bandwidth of 10Gbps, in order to allow two writes and two reads per packet. A 64-bit wide DDR interface is used, running at 133MHz, providing about 17Gbps of peak rate. With an 8-bank memory organization and random packet accesses, this corresponds to an estimated 14.5 Gbps of sustained rate, more than the minimum required.

3 INTEGRATING SCHEDULING COMPONENTS IN NETWORK PROCESSORS

3.1 The role of scheduling in network processing systems

PRO3 differs substantially from typical computing system architectures, since the notion of the main system memory does not exist nor does the system respond under the overall control of a software-based operating system. However it still resembles multi-processing architectures employed in most network processing applications and in general hardware engines broadly defined as Network Processors. Thus, the requirement though for scheduling of processing tasks (which in this case is related with packet processing) still exists. 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 TDM fashion. Process scheduling is found as a major function of operating systems that control multitasking computer systems. Packet scheduling is found in modern data networks as a mean 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 such a processor environment is used in order either to resolve contention for processing resources in a fair manner, or to process the packets/cells according to a set of traffic management rules (shaping). The scheduling problem though can not be formulated as in the case of typical processor architectures, due to the requirements of coherent protocol processing and pipeline control of the PRO3 processor (i.e. the RPMs that perform protocol processing of the “fast path” data flow). Hence no solutions that have been studied in the literature (e.g. [4], [5]) are applicable with respect to the protocol processor requirements. The same applies for the scheduling algorithms as they have been proposed and studied for traffic scheduling in data networks [6], [9]. On the other hand

there are also several commonalities between the PRO3 processor environment and the context within each of these algorithms has been studied. One first observation is that pre-emptive scheduling algorithms are not efficient since this would stall the pipelined operation of RPMs and cause extensive overhead due to context switching as has been discussed also in [5]. A second observation comes from the fact that even state-of-the-art Network Processors offer limited support for internal resource-allocation mechanisms making the problem of context switching in case of advanced scheduling schemes even worse [6]. Moreover even in state-of-the-art Network Processors the implementation of advanced packet scheduling algorithms is restrictedly resource consuming leaving very narrow headroom for the implementation of other packet processing functions [7]. Finally our motivation for developing a programmable module in hardware aiming to offload the tasks of process and traffic scheduling in an integrated way stems from the observation that the problem of task scheduling inside the Protocol Processor shares common performance requirements in terms of latency and jitter, with respect to the impact on the network traffic, with algorithms related to traffic scheduling Task scheduling differs only in the behavior and way of accessing the shared resource, which in this case is different than the case of a single network link that must be shared among multiple packet flows.

In case the RPM engines (Figure 1) process packets at a rate lower than the packet arrival rate from the network interface (IN) (i.e. when the required clock cycles for the code that has to be executed exceed the minimum packet inter-arrival interval) or in case the Insert/Extract interface is temporarily busy (bottlenecked), the packets must be stored and scheduled for processing after the target processing block has become available (i.e. a task running in software has been completed). In this case, it is possible that packets from several backlogged connections are waiting for processing, and it is desirable to distribute the processor's resources in a fair manner. Depending on the application "Fairness" may either imply low latency or guaranteed share of the resources. It is an evident that packets that belong to flows with low delay requirements (i.e. real-time, high priority traffic) should bypass the FIFO service discipline and be forwarded for internal processing with higher priority. When the processor cannot sustain worst-case conditions under line rates, such as 2.5Gbps or 10Gbps, queuing is necessary and, thus, an appropriate queuing scheme has to be implemented so as to distribute the processing resources in a way, which is consistent with several QoS criteria. Therefore, the internal scheduler must maintain a number of priority queues in order to schedule the forwarding of packets for processing according to a configurable priority per flow or per QoS class. The processing resources in the case of the PRO3 architecture, as shown in Figure 1, are the processing modules, namely the two RPMs, the on-chip RISC and the external CPU (shared via the insert/extract interface). Both the different data transactions and the communication among the internal modules, is done via the high-speed internal bus. Therefore, the arbitration of this bus is an integral part of the scheduling problem. The module implementing the internal scheduling functionality is called hereafter Task Scheduler (TSC). In the outgoing path, cells/packets processed or generated by PRO3, must be transmitted over a network link. Modern telecommunication networks require that the traffic entering the network shall comply with specific requirements so as to achieve higher link utilization, billing and QoS differentiation of services. This requirement also appears in the case that PRO3 is processes aggregated from multiple link interfaces of lower speed (e.g. 2 Gigabit Ethernet links into the PRO3 2,5 Gbps POS-PHY level 3 interface). Thus, a Traffic scheduler (TRS) is employed to shape the generated traffic according to traffic management specifications or the capacity of consequent physical interfaces.

3.2 The PRO3 scheduler architecture

According to the application requirements for a flexible lightweight scheduling twin component (both for software task and traffic scheduling), flow grouping has been employed. The performance drawbacks in the grouping case are very limited as justified and quantified in [10] and [11]. For the rest of the paper, it is assumed that packets are queued to "data queues" establishing a one-to-one association between the latter and flows. Flows are grouped to scheduling queues according to some classification rule which depends on the application /configuration. Flows are not necessarily defined as network layer flows. In the protocol processor case, one scheduler flow is bound to a specific protocol processor unit, unless load balancing feature is activated. Figure 2 shows the internal architecture of the PRO3 scheduler blocks. The Service & Redirect block (in CU) is responsible to route the commands from the Data Memory Management, the CPU and the Classifier blocks to the appropriate scheduler by means of the SID identifier kept per flow to indicate the scheduling queue for serving this flow. The data structures managed by the TSC and TRS will be called hereafter *Scheduling Queues (SIDQs)* in order to distinguish them from the per flow queues that are managed by the DMM. For the per flow queues that are managed by the DMM hereafter we will use

the terms *Data Queues(DQs)* and *flows* interchangeably.

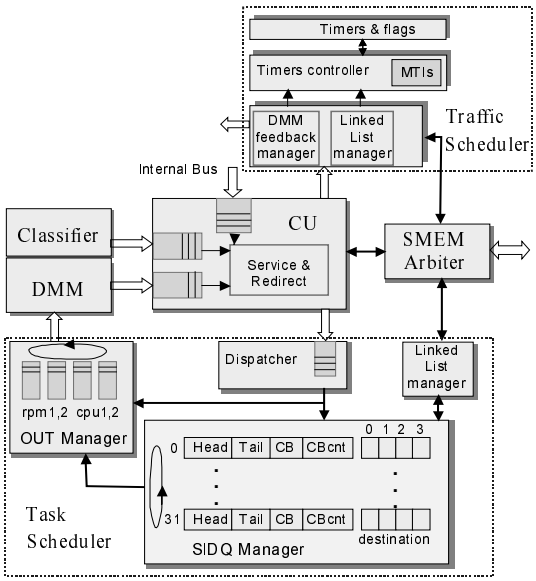

Figure 2: The Scheduler Organization

The Task Scheduler support a light, yet powerful set of commands. These are mainly used for flow management and overall control of the command path from the CPU modules to the Data Memory Manager.

SOURCE BLOCK	COMMAND	DESCRIPTION
CPU/RPM/Classifier	INIT_DQ	Initialize the Scheduling Memory entry of a specific DQ
CPU	INIT_SIDQ	Initialize Service Queue
DMM/RPM/CPU	INS_PACKET	Arrival of a new packet belonging to a specific DQ
DMM	UPDATE_QS	Update information in the scheduling memory regarding number of packets of a DQ stored in the DMM's data memory
DMM	MORE	Inform the scheduler about the status of a packet in order to continue sending more commands.
CPU	DEL_FLOW	Delete a flow
RPM/CPU	FIN_PACKET	The packet in a CPU module finished processing.
RPM/CPU	MOVE_FLOW	Move a flow from a source SIDQ to another SIDQ.
RPM/CPU	FWD_COM	Forward a command from a CPU to the DMM

Hazards may appear since it is possible for two different blocks to present commands to the Schedulers that manipulate the same flow parameters. For instance, the CPU after processing a packet decides to change the Service Queue for the corresponding DQ by means of a MOVE_DQ command. At the same time the DMM may insert a new packet belonging to this particular DQ. The CU block, that dispatches the commands, is responsible to find out the Service Queue that a DQ is assigned to; as a result it has to access the Scheduling Memory. The apparent problem of discovering the right SIDQ is alleviated by synchronizing the CU and TSC blocks to complete the critical INS_PACKET operation, before servicing the next MOVE command.

3.3 The Task Scheduler data structures organization and scheduling algorithm

The Task Scheduler supports 32 SIDQs, which are used for sharing the processing resources of PRO3 (namely the two RPMs, the on-chip RISC and the external CPU -shared via the insert/extract interface).) in a Weighted Fair Queueing manner. Each of these queues is associated with one of the possible internal destinations of packets within PRO3 and a specific protocol handler that is executed on the data of this DQ. Multiple DQs are multiplexed in one SIDQ in a Round-Robin (RR) way. The allocation of resources to the different queues is done according to some pre-configured weights. The 32 SIDQs are hierarchically organized. The first queue is treated with strictly highest priority over the others (it is mainly used to schedule traffic with low delay requirements for processing by the RPMs). The remaining 31 SIDQs can either be treated with the same priority level and be serviced in a Weighted Round Robin (WRR) fashion or (determined upon configuration) they can be hierarchically organized into two sets of 15 and 16 queues respectively with strict priority of the first set of queues over the second. SIDQs of the same priority/set are serviced in a WRR fashion.

The TSC uses a connection table (stored in the Scheduling Memory, as shown in Figure 3), where some pre-configured or run-time configurable flow parameters are stored. This table is directly indexed by the uniquely defined flowID. In this table the interesting items from a scheduling perspective terms are:

- QSTAT: The status of the DQ (empty or not)
- SID: The scheduling queue/group where it belongs
- ST: Service Time per scheduled operation/packet, used for weighted service

Entries for DQs that map to the same SIDQ are grouped in a linked list via the "Next Schedulable Flow" pointer. Apart from the above, several other parameters (supporting software) are available to the scheduler so as to initiate data transmission towards the internal destinations.

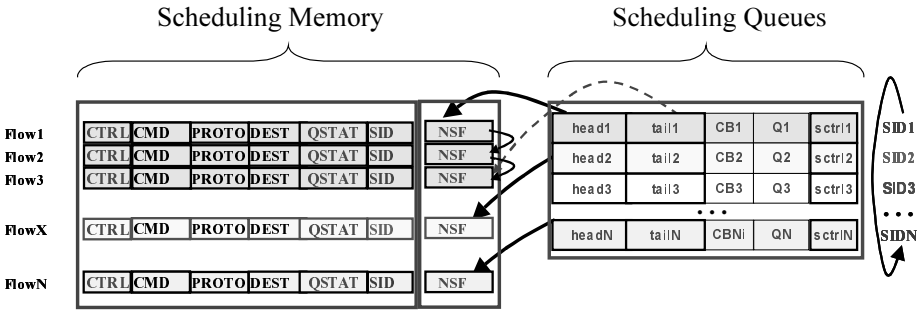

Figure 3: Scheduling Memory and SIDQ organization

The corresponding service weight should ideally be related according to the Generalised Processor Sharing approach to the portion of processing resources that will be reserved for the service of a schedulable entity. These resources mainly refer to the processing time with direct implications on the service latency. We will keep the same definition and we will denote ri the minimum service weight we want to guarantee to a scheduling queue. Real-time scheduling algorithms usually assume that the service time or the resources that need to be consumed is a-priori known. For example, the transmission of a packet of length PLi (in bytes) at line rate C will take PLi/C and requires the appropriate resources consumption. Under this assumption, if packets with low service time are processed prior to those with higher service time, the overall delay is reduced. But this is not always true: in a protocol processor environment, the service time depends on the protocol branch that will be executed. As a result, three cases exist:

1. Constant service time for all the packets of a flow, depending on the service required and the protocol executed, (e.g. plain IPv4 routing, where a few instructions are required irrespective of packet length [12])

2. The service time depends on a packet parameter (as the size) and is a-priori known (e.g. when a packet is transmitted or transferred from an input to an output port of a switch)

3. The service time is variable and only gets known after the service (as in the case of transport protocols where different protocol state machines may be executed based on the message type).

In all the cases, the desired scheduler behaviour is the maintenance of the r_i as defined in Equation 1 (where $S_i[I,\tau]$ is the system processing time allocated to flow i by unit I within an interval τ, in which all flows remain backlogged). In this direction the configurable service time (ST) parameter represents the service weight. Since the case of the protocol processor in most applications matches case 3 discussed above and in order to achieve high operation speed, the implementation of weighted round robin algorithm, which has a complexity of O(1) and can achieve high throughput under all conditions was selected for the PRO3 implementation. A simple but adequate implementation way for the WRR algorithm is described in the piece of pseudo-code in Figure 4.

```
(* Initialization *)
for ( i = 0; i < SIDq; i++ )
   CB_i = ST_i ;

(* Weighted service discipline *)
for ( i=0; i<SIDq; i++ )
   if ( SIDQ_i NOT IDLE)
      h_flowID_i = Head(SIDQ_i);
      Remove(h_flowID_i, SIDQ_i);
      if (CB_i > 0 )
         Service(h_flowID_i);
         if (h_flowID_i NOT IDLE)
            Insert(h_flowID_i, SIDQ_i);
         CB_i = CB_i - 1;
         RETURN; (* exit for loop and start over *)
      else
         CB_i = ST_i;
```

Figure 4: simple WRR implementation pseudo-code

```
(* Initialization *)
for ( i=0; i<SIDq; i++ )
   CB_i = 0;

(* Weighted service discipline *)
for ( i = 0; i < SIDq; i++ )
      if ( SIDQ_i NOT IDLE)
         h_flowID_i = Head(SIDQ_i);
         Remove(h_flowID_i, SIDQ_i);
         Srv_Time_i = ST(h_flowID_i);
         CB_i= CB_i+Q_i;
         if (Srv_Time_i ≤ CB_i)
            Service(h_flowID_i);
            CB_i = CB_i - Srv_Time_i;
         if (h_flowID_i NOT IDLE)
            Insert(h_flowID_i, SIDQ_i);
```

Figure 5: DRR implementation pseudo-code

$$r_i = \frac{S_i[I,\tau]}{\sum_{i=1}^{n-1} S_i[I,\tau]}$$

Equation 1: Reserved share of system resource *I* for flow *i* in time interval τ

$$r_i = \frac{\sum_{j=1}^{n-i} \prod_{k=i}^{n-j} ST[k]}{\sum_{i=1}^{n-1} \sum_{j=1}^{n-i} \prod_{k=i}^{n-j} ST[k]}$$

Equation 2: Minimum share of service opportunities of SIDQ i (when all queues remain backlogged)

The above algorithm guarantees that the number of visits and service opportunities of each SIDQ is proportional to their relative order of priority and has simpler implementation for the restricted number of SIDQs than that described in [13]. The algorithm described in Figure 4 does not follow the approach of nearly equally spaced in time visits of the N queues but exhaustively serves highest priority queues (in a decreasing order of priority) up to their maximum share per "round". In this case the notion of "round" is determined by the (maximum) time that elapses between two consecutive visits of the latest SIDQ. Specifically the minimum share of service opportunities of the ith SIDQ among the n in total is given by the formula of Equation 2.

Since the processing time is not fixed in all applications and may vary for different packets/protocols the fixed slot time division of WRR implementations may not be adequate. The equivalent of Weighted Round Robin (which is applicable in fixed, one-slot duration service time operations like in the case of scheduling ATM cells) for packet level and generally variable time duration operations is an algorithm called Deficit Round Robin (DRR) [14], the main algorithm of which we repeat in Figure 5. It is evident that the DRR discipline, which yields also efficient

implementation in hardware, can also be supported by the scheduler design described above, utilizing the same data-structures and slightly modified service FSM (initialize at 0, count up by Qi and update CBi, until CBi greater than STi). The data organization described in Figure 3 can be used for implementing the Deficit Counter (DC) (in the place of the generic "Balance" field) and allocated Quantum (Q) (in the place of the generic Assigned Value field) as used in [14].

In the case of the protocol processor the ST parameter is generally dependent on the type of protocol to be executed and consequently the type of message and less on the message length. The ST parameter can then be statically configured per flow. Thus flows grouped into the same SIDQ shall have a common destination and allocation of internal resources but may have different ST parameters that may represent the processing resources (slots per packet) required for different handlers/protocols. For the traffic scheduler, which defines the packets transmission order, the STi parameter is proportional to the packet length and equals Li/C (where Li the packet length and C the link rate). The length of each individual packet is stored in the pointer/context information by the Data Memory Manager, which actually handles the packet pointers and related information. The DMM, after a service command has been issued by the schedulers, shall update the ST parameter of the selected flow with the packet length of the next pending packet. This has some limiting implications since a) it requires an update of the connection memory per slot (in worst case); thus deteriorates the pointer access overhead, and b) the round-trip time required for a correct update of the flow context grows larger due to the packet pointer accesses, limiting the maximum rate that can be allocated to a single flow (in case it is the only one that occupies the system).

3.4 Internal resource management by means of the Task Scheduler functionality

3.4.1 Load balancing

The Task Scheduler treats the RPMs that perform protocol processing of the "fast path" data flow in an equal manner. Each flow may be assigned to either one, or to both if load balancing is enabled. The RPM blocks indicate directly to the Task Scheduler the availability of their incoming FIFOs. It is therefore easy for the TSC to decide whether an eligible flow that is programmed to "both" RPMs should be directed to the non-busy one. In addition, the output FIFOs inside the Out Manager of the TSC (see Figure 2) are also taken into account. However, the consequence of pipelined operation of RPM may result in an awkward effect: consequent packets from a flow initially scheduled to RPM1 may be again scheduled to RPM. This should be suspended in order to achieve coherent protocol processing and in-order flow state accesses to the control RAM. Due to the RPM proprieties and their asynchronous processing this control function (equivalent to the *MUTEX* software technique) could not be efficiently dealt with even in software (e.g. by means of semaphores). This is alleviated by allowing the Scheduler to modify the destination field of one flow that is assigned to "both" RPMs upon taking the decision to transmit it to the one currently available. In the same time it is marked. Hence, if the processing of the packet from this flow finishes before this flow becomes eligible again, the original destination field is restored; otherwise the RPM chosen the first time will also process the consequent packets.

3.4.2 Pipelined operation control

Since some applications may not allow the pipelined operation of RPM either for consecutive packets of a single flow or generally from any flow, TSC is also designed to disable pipelined packet forwarding by means of both on- and off-chip flags that (re)set the schedulability of either each DQ or each SIDQ seperately. On-chip flags are used to disable pipelined operation of a specific destination. In the per-flow context respectively a disable flag exists in each flow entry is used and set after a service, when the respective flags indicate that pipelined operation is prohibited by the application programmer. For this purpose the FIN_PKT(flowID) command is supported and must be issued by the internal module in order to reset the Disable flag and enable the scheduler after the processing of previous packets is completed.

3.4.3 Internal bus sharing and split-transaction support

TSC is also used to multiplex the execution of data transactions from the DMM to the different internal destinations via the internal bus and allow for interleaved transactions over the IBUS. For this purpose all modules that request the

execution of an operation from the DMM that fetches data from the external memory over the internal bus must direct those commands to the TSC. TSC orders the pending commands per destination module and issues interleaved commands based on a configurable default size of data segments. To perform this task the TSC interprets DMM commands that result in variable length execution times and issues multiple commands to the DMM for a default size of data. This gives the opportunity to the ARB to grant in the meantime of the execution of a complex operation access to the IBUS for another transaction. Also in case a specific destination becomes bottlenecked and raises its busy signal the TSC will stall execution of commands targeting this destination and continue servicing commands (if any) sending data to the non-busy internal destinations.

3.5 The Traffic Scheduler algorithm

The outgoing Traffic Scheduler (TRS) performs peak-rate shaping of outgoing ATM cells or IP packets. Each flow's peak rate (PR) is compiled to the Minimum Transmission Interval (MTI) through Equation 3, where PDS is the predefined data segment size, which in the ATM case is 53 bytes while in the IP case coincides with the segment size supported by the DMM. TRS implements 32 SIDQs in total, each associated with a certain peak-rate and utilizes the same data structures as TSC. 32 discrete rates are deemed adequate [9] for a line rate of 2.4Gbps and the rate resolution becomes higher as peak rate decreases, (Equation 4).Table A presents the maximum and minimum rates supported for different TRS configuration under IP and ATM applications.

$$PR = \frac{PDS}{(MTI + 1)slot}$$

Equation 3: Packet shaper Peak Rate calculation

$$Granularity = \frac{PDS}{(MTI + 1)^2 slot}$$

Equation 4: Packet shaper rate granularity

Application	Max Rate (Gbps)	Min Rate (kbps)
IP Firewalling	1	153.6
Single Phy ATM	0,622	38.8
Multi Phy IP/ATM	2.4	386.6

Table A: TRS rate granularity

For each SIDQ a countdown timer is kept. This is set to the MTI value, is decreased by one after one "slot" time - where "slot" time is the time required for the transmission of PDS bytes- and its expiration triggers the service of each active flow classified to this SIDQ. In ATM operation, AAL packets leave the system cell by cell. Each time that the timer of SDIQ I expires, all the active flows of this queue are serviced, i.e. one cell from the corresponding data queues is transmitted. In IP applications, when the timer expires the transmission of packets is commanded to the DMM, which responds with the packet size measured in PDS. The TRS then does not service an equal number of timer expirations. This is equivalent to setting the timer to the value MTI*(PL/PDS), where PL is the packet length i.e. TRS supports aggregate per group peak rate shaping for IP flows whereas peak rate shaping per ATM flow can be guaranteed.

4 SCHEDULER ARCHITECTURE EXTENSIONS AND DIMENSIONING

One important feature of the scheduler architecture described in section 0 is its flexibility to be extended for other Network Processor architectures and multiple traffic management mechanisms in an efficient way. The PRO3 architecture is a single-in, single-out architecture; therefore the dual scheduling component is deemed enough to support multiplexing of flows on these two ports. The internal scheduling paradigm though with the multiple destinations can be re-used for implementing a class of scheduling algorithms, including those described in section 3.3, for scheduling packets to be transmitted over multiple network interfaces. An extension of the Network Processor architecture that could exploit these traffic management extensions is shown in Figure 6. 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 in section 0 to support multiple numbers of virtual and physical output interfaces as shown in Figure 7.

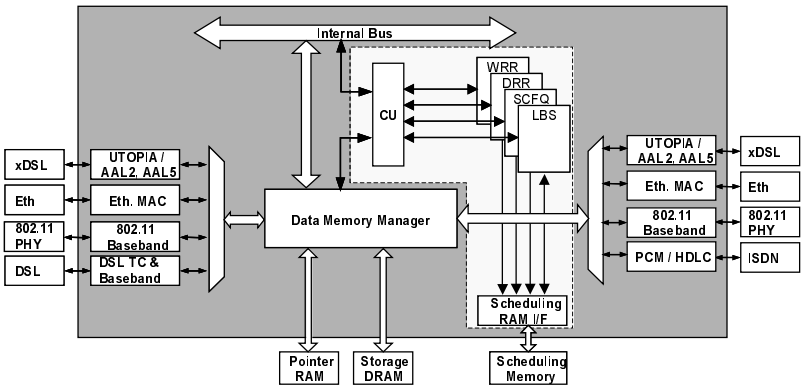

Figure 6: Architecture extensions for programmable service disciplines

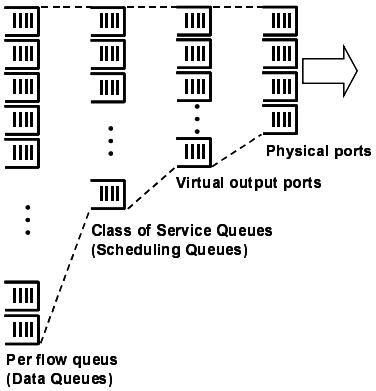

Figure 7: Queuing requirements for multiple port support

The PRO3 scheduler, as described in section 0, inherently supports these hierarchical queuing requirements by means of independent scheduling per DQ, SIDQ 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 SIDQs per CoS, we can achieve very efficient virtual and physical port scheduling. Implementation of more scheduling disciplines can also be achieved easily, by simply adding the service execution logic (Finite State Machine - 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 that is required is the extension of the CU and memory controller modules to support a larger number of ports. The required throughput of the Pointer Memories used still remains the same as long as the aggregate bandwidth of the incoming network interfaces is at most equal to 2.5Gbps that the DMM can handle. The only limitation is related to the number of supported SIDQs, 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 SIDQ memories and priority enforcers for fast searching in these memories, which we can efficiently extend to very high numbers of SIDQs as presented in [15]. In addition, functionality already present in the current scheduler implementation allows for deferring service of one SIDQ 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 Network Processor 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). The efficient implementation of packet fair queuing algorithms like SCFQ, according to the generic methodology presented in this paper has also been discussed in [16]. Moreover a novel feature of our architecture is its flexibility to implement hierarchical scheduling schemes without necessitating data movement but only pointer movement. For example multiple packet flows that traverse an IP/ATM network path can share ATM Virtual Paths or Circuits (VPs, VCs) by means of a packet fair queuing algorithm, eligible packets can be appended to ATM VC queues and scheduled by means of WRR to share the outgoing ATM link. Another use of scheduling flows via multiple schedulers is the support of Available Bit Rate (ABR) service in ATM networks, which can easily be achieved by assigning DQs to both SIDQs, one for minimum rate scheduling and to a second for best effort RR scheduling to share the excess bandwidth. 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 issue that needs to be tackled in hardware is the handling of busy indication signals from the different physical ports to determine schedulable flows/SIDQs.

5 CONCLUSIONS

This paper focused on designing efficient, flexible and scalable scheduling components for network SoCs used for protocol processing in Gigabit networks. The scheduler architecture has been validated against the architecture of a reference network SoC. The intricacies of task and traffic scheduling inside the PRO3 protocol processor architecture have been outlined and requirements at the system level have been analysed. The architecture of the scheduling components yields efficient VLSI implementation, with low memory requirements and flexibility to support multiple service disciplines in a programmable way, supporting thousands of flows. The design is suitable both for cell and packet based network-processing applications and has been efficiently implemented in VLSI, occupying 1.7 square microns in a 0.18um technology. An extension of the PRO3 scheduler architecture has been proposed to efficiently support integration of scheduling components as programmable hardware cores in multi-port Network Processor architectures with hierarchical queuing and scheduling requirements. The proposed architectural enhancement retains the inherent features for integrated internal resource management and traffic management, while it can achieve significant performance enhancements for single-chip Network Processor implementations.

REFERENCES

[1] K. Vlachos et al. *"Processing and Scheduling Components in an Innovative Network Processor Architecture",* 16th IEEE International conference in VLSI design, New Delhi, India, January, 2003.

[2] Ch. Ykman-Couvreur et al. *"System-level performance optimization of the data queueing memory management in high-speed network processors"*, In Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June, 2002

[3] Hyperstone Electronics E1-32X RISC/DSP, http://www.hyperstone.com.

[4] R. Gopalakrishnan, Gurudatta M. Parulkar, *"Bringing Real-Time Scheduling Theory And Practice Closer For Multimedia Computing"*, in Proceedings of ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Philadelphia, United States, 1996

[5] K. Ramamritham, J. A. Stankovic, *"Scheduling algorithms and operating systems support for real-time systems"*, IEEE proceedings, vol.82, no.1, pp.55-67, January 1994.

[6] A. Srinivasanm et al. "Multiprocessor Scheduling in Processor-based Router Platforms: Issues and Ideas," Workshop on Network Processors, 9th International Symposium on High-Performance Computer Architecture, Anaheim, California, February 2003.

[7] I. Paul, W. Shi, X. Zhuang, K. Schwan, *"Efficient Implementation of Packet Scheduling Algorithm on high-Speed Programmable Network Processors"*, In Proceedings of the 5th IFIP/IEEE International Conference on Management of Multimedia Networks and Services, MMNS 2002, Santa Barbara, CA, USA, October, 2002.

[8] A. Varma, D. Stilliadis, *"Hardware Implementations of Fair Queuing Algorithms for Asynchronous Transfer Mode Networks"*, IEEE Communications Magazine, pp 54-68, December 1997.

[9] D. C. Stephens, J. C. R. Bennett and H. Zhang, *"Implementing Scheduling Algorithms in High-Speed Networks",* IEEE Journal on Selected Areas in Communications, Vol. 17, pp. 1145-1158, June 1999.

[10] S. Zeng, N. Uzun, *"A Hierarchical Traffic Shaper for Packet Switches"*, Globecom 1999, Rio de Janeiro, Brazil, December 1999.

[11] J. Rexford, F. Bonomi, A. Greenberg, A. Wong, *" Scalable Architectures for Integrated Traffic Shaping and Link Scheduling in High-Speed ATM Switches"*, IEEE Journal on Selected Areas in Communications, pp 938-950, June 1997.

[12] D. Clark, V. Jacobson, J. Romkey, H. Salwen, *"An Analysis of TCP Processing Overhead"*, IEEE Communications Magazine, June 1989.

[13] M. Katevenis, S. Sidiropoulos, C. Courcoubetis, *"Weighted Round Robin Cell multiplexing in a general-purpose ATM switch chip"*, IEEE Journal on Selected Areas in Communications, Vol. 9, No 8, October 1991.

[14] M. Shreedhar, G. Varghese, *"Efficient Fair Queuing using Deficit Round Robin"*, IEEE Transactions on Networking, Vol.4, No 3, June 1996.

[15] G. Kornaros, Th. Orphanoudakis, I. Papaefstathiou, "Active Flow Identifiers For Scalable, Qos Scheduling" ISCAS'03, 2003 IEEE International Symposium on Circuits and Systems, May 25-28 2003, Bangkok, Thailand

[16] J. Rexford, A. Greenberg, F. Bonomi, "Hardware-Efficient Fair Queueing Architectures for High-Speed Networks", Infocom, San Francisco, CA, USA, March 1996