DESIGN AND EVALUATION OF A BURST ASSEMBLY UNIT FOR OPTICAL BURST SWITCHING ON A NETWORK PROCESSOR

Jochen Kögel, Simon Hauger, Sascha Junghans, Martin Köhn,
Marc C. Necker, and Sylvain Stanchina
University of Stuttgart
Institute of Communication Networks and Computer Engineering (IKR)
Pfaffenwaldring 47, 70569 Stuttgart, Germany
{koegel, hauger, junghans, koehn, necker, stanchina}@ikr.uni-stuttgart.de

Abstract Optical Burst Switching (OBS) has been proposed in the late 1990s as a novel photonic network architecture directed towards efficient transport of IP traffic. OBS aims at cost-efficient and dynamic provisioning of sub-wavelength granularity by optimally combining electronics and optics. In order to reduce the number of switching decisions in OBS core nodes, traffic is aggregated and assembled to bursts by the Burst Assembly Unit in an OBS ingress edge node. This Burst Assembly Unit is responsible for buffering incoming packets in queues and sending them as bursts as soon as a minimum burst length is reached and/or a timer expires. Typically, dozens of different queues must be able to handle high volumes of traffic.

This paper presents the design and implementation of a Burst Assembly Unit for a Network Processor. In an evaluation of the realized implementation we point out the ability to handle traffic at line speed while having fine grained timers for all queues.

Keywords: Optical Burst Switching, Traffic Aggregation, Network Processor, System Design

1. Introduction

Optical Burst Switching has been introduced as a new switching paradigm for transport networks in order to compensate the two main drawbacks of todays network architectures [Qiao, 1999]. First, today's optical transmission technology is only used for point-to-point links between network nodes, while switching and routing is done in the electrical domain. Thus, all data has be converted optical-to-electrical (O/E) and electrical-to-optical (E/O) in every

node. With optical switching, data would always remain in the optical domain and thus O/E and E/O conversion would only be necessary on the edge. Second, transport networks either rely on the packet or on the circuit switching principle. In packet switched networks, statistical multiplexing gain can be realized leading to a high efficiency, but at the cost of a high processing overhead and the need for buffering. In circuit switched networks, in a core node only simple operations are required to forward the incoming data stream to the responsible output port without large buffers, but it cannot capitalize on any statistical multiplexing. In OBS, packets are collected in the edge nodes and assembled according to a certain strategy [Dolzer, 2002] into bursts with a size usually between 10 kilobit and some 100 kilobits [Gauger, 2003]. These bursts are sent through the core network to the egress node remaining always in the optical domain. Here, the bursts are disassembled and the packets are forwarded towards their destination. With this, still statistical multiplexing can be used to increase the efficiency of the network, while the processing overhead is essentially reduced in relation to packet switched networks. Furthermore, O/E and E/O conversion is necessary only at the edge of the network.

In an OBS edge node, the task of the Burst Assembly Unit (BAU) is to classify incoming packets, buffer them in the corresponding queues, assemble the bursts, and finally schedule and transmit them. A BAU could be implemented on several types of hardware: ASICs, FPGAs, Network Processors (NP) or even General Purpose Processors (GPP). In contrast to a GPP, an NP is equipped with fast media interfaces and optimized for parallel and pipelined processing as it is typical for network applications. Compared to hardwired solutions, network nodes using NPs can be adapted to new requirements by simply changing the program. Most NPs are designed for classical IP processing applications consisting mainly of packet classification, forwarding, queueing and scheduling. Further, the manufacturers provide software frameworks optimized for applications based on IP and Ethernet. Nevertheless, in a BAU several new aspects like creating new packets or multi stage queueing become important. These new tasks are a challenge to the NP's flexibility.

This paper presents the design of a BAU for an Intel IXP2400 Network Processor. We will show how the BAU has been realized by a Burst Assembly Module (BAM) that has been integrated in the manufacturer's software framework.

The remainder of this paper is structured as follows: Section 2 introduces OBS and Burst Assembly, Section 3 describes the Network Processor used. In Section 4, the design of the BAM is presented, which is integrated in the BAU in Section 5. The results of the performance evaluation are presented in Section 6. The last section concludes the paper.

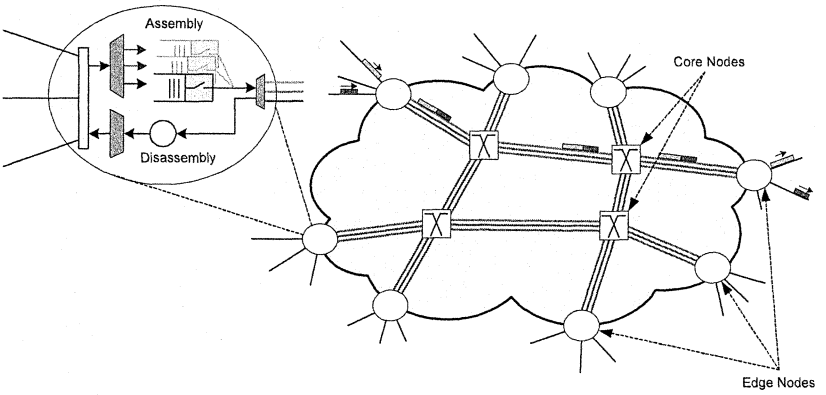

Figure 1. OBS Network

2. Optical Burst Switching

Current network architectures use fibers with several wavelengths for point-to-point links between nodes. On each node, data has to go through O/E conversion for switching and through E/O conversion thereafter. In OBS, data remains in the optical domain throughout the network.

Figure 1 shows an OBS-network. It consists of edge nodes, core nodes and fiber bundles connecting the nodes. The edge nodes aggregate traffic to bursts, which are then sent through the core to the egress node. Arriving at the egress node, the Disassembly Unit extracts the packets from the burst and forwards them to their destination.

A closer look on the Burst Assembly Unit in Fig. 2 reveals its functional blocks. First, received packets are forwarded according to egress node and

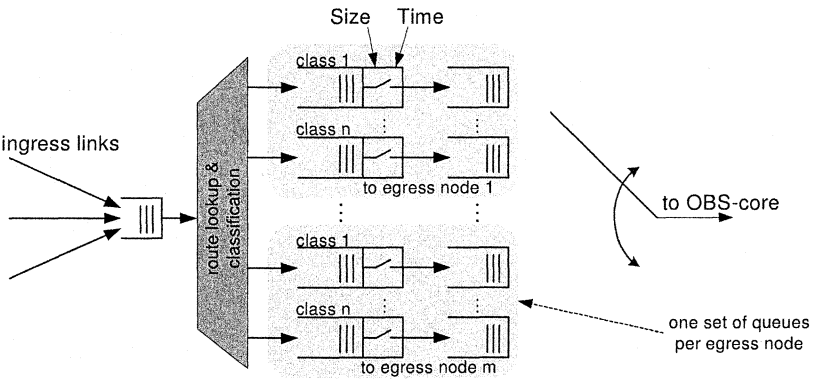

Figure 2. Burst Assembly Unit for n traffic classes and rn egress nodes

traffic class to one of n queues of the m sets of queues. Once finalization is triggered, all packets of a queue are assembled to a burst and forwarded to the next queue. On finalization, a header is generated containing information on the positions of the packets within the burst, needed for disassembly on the egress. After finalization, the burst is queued in another queue before it is scheduled for transmission to the OBS core. Finalization can be triggered based on the following criteria:

- **Size-based**
  Once the queue length reaches a minimum size.
- **Time-based**
  A predefined time after the first packet has been enqueued.
- **Combined**
  Reaching the minimum size or timeout trigger finalization.

In the time-based case, the maximum delay of a packet due to queueing is determined by the timeout set. Furthermore, for providing several levels of Quality of Service (QoS) for different traffic classes, the size/timeout values can be set accordingly and appropriate scheduling can be employed [Vokkarane, 2002].

Due to properties of the OBS network, the maximum burst size might be limited. If this is the case, finalization has also to be triggered in cases where a burst would become too large by enqueueing another packet, since neither size-based nor time-based strategies assure a maximum burst size.

3. Network Processors

3.1 Overview

Processing tasks in network nodes can be classified in data path and control path tasks: While the majority of packets has to be forwarded through the node on the data path, some packets are addressed to the network node itself for network control and routing. For the data path, Network Processors integrate several Special Purpose Processors (SPP) optimized for forwarding large amounts of data. For control path processing and management tasks, a General Purpose Processor (GPP) is often included.

The architectures of the SPPs are optimized for the characteristics of the data path: Operations like table lookup or queueing are supported by special units and several SPPs do processing tasks either as a functional pipeline, in parallel, or in a combination of both. In network nodes, packets have to be read and written at high speed, while only the header, a small part of this data, is processed. Therefore, and because the header has to be processed only once, there is no temporal or spacial locality of this data and thus, caching of data

is not useful. However, the processor should not waste processing time while waiting for memory operations. Since no cache can hide this memory latency, other methods, like hardware-supported multithreading, are employed.

The common processing approach in network nodes is to first do packet processing (e.g. forwarding), followed by a queueing stage. Afterwards, the scheduler decides when a packet should be transmitted. Following this pattern, some NPs reflect this principle in their hardware design: A cluster of SPPs for packet processing and a dedicated "Queue Manager" (QM) for queueing. Often, these two components are implemented in separate chips with different hardware requirements: While the QM needs a high amount of memory for packet buffers, the SPPs require low latency memory for tables and a small amount of packet buffer memory only.

Different from this common principle, for Burst Assembly two stages of queueing are needed: First, the packets have to be queued until a burst is finalized and second, the burst is queued until it will be scheduled. Obviously, an NP that does not need an external QM component is better suited for this kind of application than a two-chip solution. For the implementation of the BAU, an NP of Intel's IXP2XXX series has been chosen, since these processors do not adhere to the division of NP and QM.

3.2 The IXP2400

There are five different NPs in Intel's IXP2XXX series. All of them comprise of the same type of SPPs (called microengines) for data path processing. The different models of the IXP2XXX series differ in the number of microengines employed, as well as in memory size and clock frequency of microengines. For our implementation, an IXP2400 has been chosen, which contains eight microengines and consists of the basic blocks that are shown in Fig. 3.

Microengines are simple 32-Bit RISC processors that allow hardware multithreading by supporting fast context switches between the eight register sets. In addition to the registers that are intended for per-packet data, local memory within each microengine can be used to keep state information accessible for all threads. All microengines have access to the media interface, the external memory and to a fast on-chip memory. While the external memory is intended for packet buffers and lookup tables, the on-chip memory is used for inter-microengine communication. Additional hardware support for ring buffers (FIFO queues) allows for atomic concurrent access to this shared on-chip memory. Since all microengines can access the same units in the same way, the tasks of the packet-processing pipeline can be mapped onto the microengines in either way, parallel or pipelined.

A software framework [Intel, 2003] is provided by the manufacturer, which contains building blocks for common processing tasks and defines data struc-

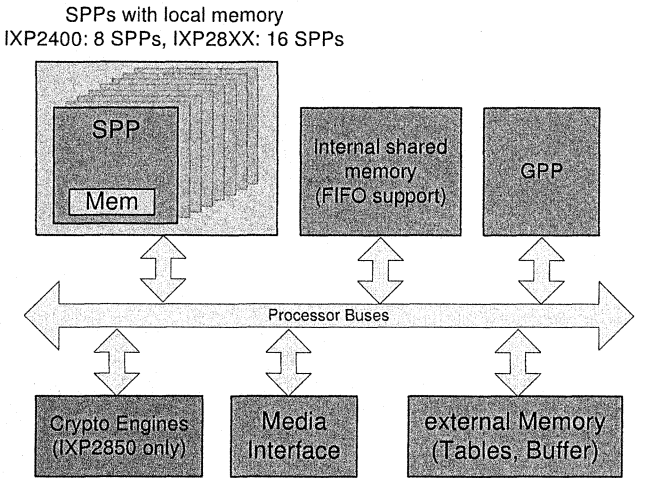

Figure 3. Simplified architecture of the IXP2XXX

tures as well as interfaces. This framework enables designing reusable and flexible software blocks for all models of the NP series.

3.3 Mapping of functions to microengines

The software framework defines one possibility of mapping processing functions to microengines. In the following we will introduce the framework briefly, since building blocks of it have been reused for the BAU.

Figure 4 shows from top to bottom the basic tasks of a packet processing node, the decomposition of the tasks in basic functions, and the mapping to microengines as defined by the software framework. This functional pipeline works as follows: The RX and TX blocks transfer packet data from the media interface to DRAM and vice versa. For passing the packet from block to block, only a pointer to the packet (packet handle) is used. This handle is passed together with other information between microengines by the use of FIFOs residing in the shared on-chip memory.

The packet processing stage incorporates all processing on a packet and its headers, like decapsulation, encapsulation, validation, forwarding and classification. Since this stage needs the most of both, processing power and memory access, it runs in parallel on several microengines. The queue and scheduler blocks each run on separate microengines and are interconnected by FIFOs for transferring control messages, not packet handles. Once a packet is scheduled, its handle is forwarded to the TX stage, which can run in parallel on two microengines for performance reasons.

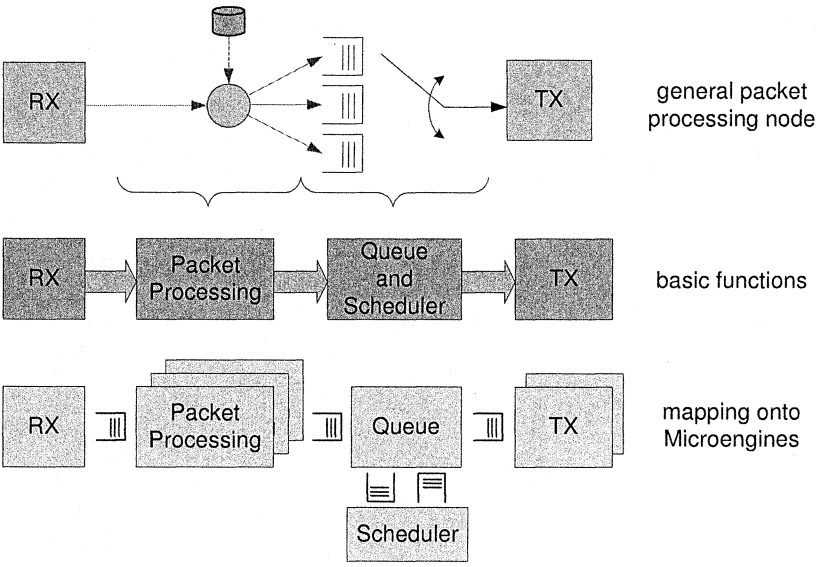

Figure 4. Mapping of functions to microengines as defined by the software framework

4. Design of the Burst Assembly Module

For queueing of packets and assembling bursts, a software module has been implemented that is designed for running on a single microengine using the eight hardware-supported threads. Other functional blocks, necessary for a complete BAU have been realized by the reuse of existing building blocks. The BAM can be integrated in the functional pipeline defined by the software framework (section 3.3), as it will be shown in Section 5.

4.1 Basic tasks

As shown in Fig. 5, the BAM receives packets from the input FIFO and sends bursts by writing to the output FIFO. In addition, it is capable of prepending headers to bursts.

For storing packets, the software framework defines a buffer structure of a fixed size. In the case of large packets, several buffers can be concatenated by means of pointers resulting in a linked list. Converting this multi-buffer packet back to a contiguous packet is done at the end of the pipeline by the TX-block, which reads all buffers of such a list and sends the data to the media interface.

The BAM uses this support for linked lists to build chains of packets. These chains are used for both, queueing of packets, as well as for sending the burst to downstream blocks, which will handle the linked list as if it was a single

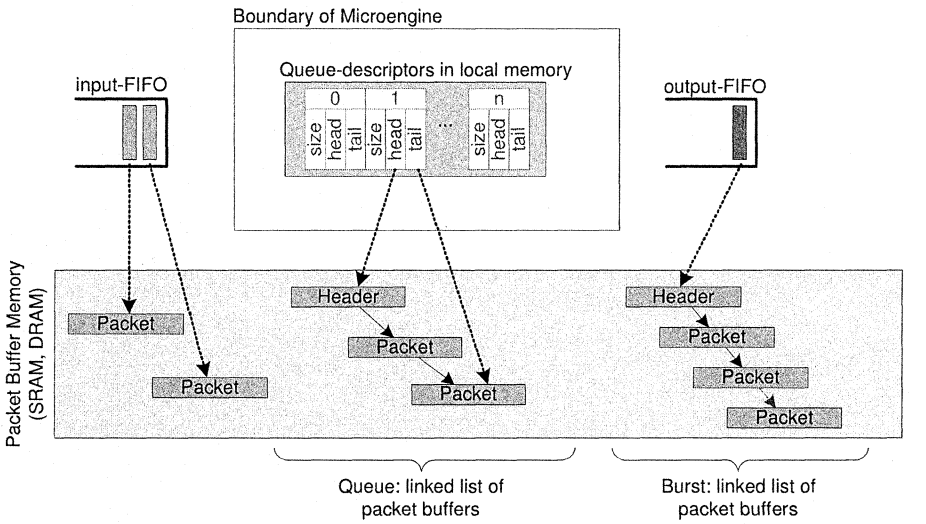

Figure 5. Queueing and assembly using linked lists of packet buffers

multi-buffer packet. The chains are managed by queue descriptors (see Fig. 5), which keep a pointer to the head and tail of the queue as well as its size.

For enqueueing packets, a data structure from the input FIFO is read, which contains the packet handle and information on the packet size and queue number. The queue number is determined by upstream blocks and based on its value, a queue descriptor is selected and the packet is enqueued to the tail of the corresponding queue. Before the first packet is added to a queue, a buffer for the header is allocated and enqueued, which provides enough memory space for a complete burst header.

Finalizing a burst is accomplished by writing a data structure to the output FIFO and resetting the queue, which is done by setting its size to zero, and enqueueing a new header buffer. The data structure written to the output FIFO contains the handle of the first packet and the queue number. This handle contains sufficient information for downstream blocks to have access to the complete burst, while the queue number is needed since the next stage is a queueing stage that uses the same queue numbering as the BAM. Triggering the finalization of a burst for the size-based case is simply done by checking the size of the queue, while for time-based assembly, a timer mechanism is employed, which will be explained in section 4.2.

The burst header contains one field for the number of packets, one for the size of the header and one for the size of every packet. Obviously, the size of the header is not fixed and is only known when the burst is finalized.

For performance reasons, the header is written in chunks of eight bytes. Since each header field has a size of 16 bits, four packet size fields have to be buffered within the microengine before they are written to the buffer. When the burst is finalized, the remaining buffered size fields are written.

4.2 Timers

As seen above, timers are necessary for time-based assembly for every queue. A timer is started when the first packet for a new burst is received and disarmed when the burst is sent. When a timer expires, the corresponding burst is finalized.

Although there are eight hardware timers per microengine, they are not used since they cannot be disarmed and it is rather difficult to map about dozens of timers to eight hardware timers efficiently. Therefore, a timer mechanism based on polling of a timestamp register has been realized.

This timer mechanism saves the target times, at which the timer expires, in local memory and checks them periodically. For the target times, a contiguous block of memory is reserved, with each timer occupying a 32-bit word. For efficiency reasons, processing the timers is distributed among all threads with each thread handling up to eight timers. Thus, with eight threads, up to 64 timers can be processed. This implementation is very fast, because eight contiguous memory words can be accessed very efficiently with this hardware and thus needs less processing time than an implementation with longer lists or calendars. For keeping the program simple, each thread processes the same number of timers. Hence, the number of timers is always a multiple of eight, which may result in spare timers that remain always disarmed.

A system time, which is provided by a 32-bit timestamp register in each microengine, is used for this timer scheme. The system clock increments this timestamp every 16 processor cycles, which is obviously too fine-grained for our needs. Therefore, the desired granularity is achieved by executing a right shift operation on this timestamp value, thus the granularity can be adjusted easily. For reducing jitter, the internal granularity for saving target times in memory is four times finer than the timer's granularity.

When designing a timer, which relies on an advancing system time, overflows of this value have also to be considered. This can be accomplished in several ways, which all introduce additional checks and expensive branch instructions. This design circumvents overflow handling by only checking for equality of system time and target time instead of doing a less-or-equal check for determining exceeded timers. However, in this case, it has to be guaranteed that all timers can be processed fast enough before the time advances. This condition has to be checked for a desired timer granularity and if it does not hold, a coarser granularity has to be chosen.

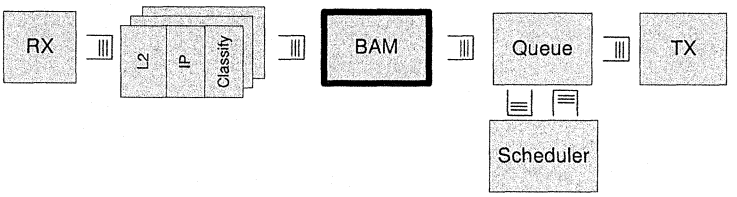

Figure 6. Integration of the BAM into the BAU

5. Integration and Test

For a complete BAU, the BAM has been integrated in the functional pipeline between packet processing stage and queue (Fig. 6). This functional pipeline reflects the BAU introduced in Fig. 2. All other blocks can be realized by the use of existing building blocks with minor changes.

The BAU has been realized on a Radisys ENP-2611 Network Processor Board, which is equipped with an IXP2400, three optical Gigabit Ethernet ports, and various other supporting components (e.g. memory for packet buffers and tables). The Ethernet ports are connected to the media interface of the IXP2400.

Using this hardware, it is possible to test the BAU as part of a network scenario, where IP packets are received and bursts are sent over Ethernet. Thus, in contrast to future OBS applications, the burst size is limited for this implementation to 9000 bytes (Ethernet jumbo frames).

As shown in Fig. 7, the test setup consists of the NP board and two computers, working as source and destination node. On the destination node, a disassembly program extracts packets from the received bursts and forwards them to the local network stack. In order to acknowledge test packets received, a feedback link is set up directly between the two test nodes. On the destina-

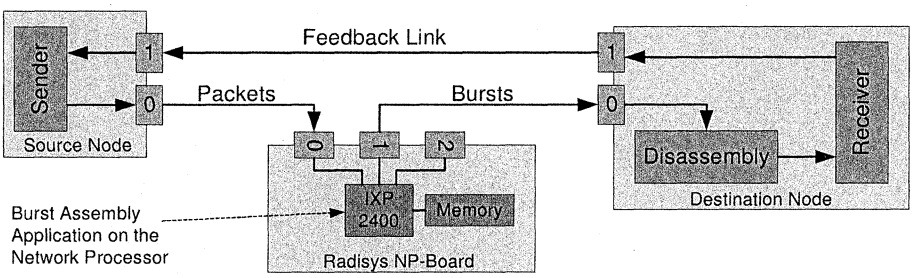

Figure 7. Setup for testing

tion node, a receiver software for responding to test packets is running, which can be an ICMP stack for simple tests.

Using this setup, the functionality of the BAU could be validated and a throughput of 500 MBit/s was achieved, using [Iperf] as measurement tool. A higher bandwidth could not be achieved due to limitations of the computers used as source and destination. However, as the next section will show, the BAM is capable of handling traffic rates that are higher than the interface speed. Thus, it is impossible to measure the BAM's maximum throughput using the test setup described.

6. Evaluation of the BAM

In this section, the scalability and performance of the BAM is evaluated. Other pipeline stages and their impact on the performance are not considered.

6.1 Scalability

In the following, the scalability limited by the amount of microengine-internal memory will be discussed. The memory requirement per queue depends on the desired assembly strategy and on whether a header should be prepended to the burst or not: In equation (1) M, the amount of local memory needed, is determined by the number of queues (n_q), the size of the queue descriptor (s_{qdesc}) and the memory needed for one timer (s_{timer}), which is always 4 bytes. For size-based assembly without header generation, s_{qdesc} is 16 bytes, while it is 32 bytes for all other cases.

$$M = n_q \cdot s_{\mathrm{qdesc}} + \left\lceil \frac{n_q}{8} \right\rceil \cdot s_{\mathrm{timer}} \tag{1}$$

Since the microengines of the IXP2400 have 2560 bytes of local memory each, 64 queues can be handled if all features are used. This number would be enough for a wide area network (WAN) of the size of Germany for example. Such a network might have about 30 edge nodes and for service differentiation, two traffic classes could be employed, which would result in about 60 queues per BAU. For more queues, external memory had to be employed resulting in more complex and slower design.

6.2 Performance Estimation

In this section, we will show how the BAM has been evaluated by calculations based on the worst case cycle count. The results are presented for different configurations of the BAM.

In order to calculate the number of packets that can be processed per second, the proceeding is as follows: First, the number of cycles needed for the

various tasks of burst assembly has to be counted and second, it has to be calculated how many of these tasks can be processed within a given period of time, resulting in the number of packets processed.

Calculating the throughput based on cycle count is possible due to the properties of the microengine's architecture: The microengines are based on a RISC architecture with a single execution path. Therefore, one instruction is processed in one clock cycle. Additionally, the integrated instruction memory is fast enough to read the next instruction within one clock cycle, hence there is no latency for reading program data.

The microengines are programmed in an assembly language using the instructions that will be executed on the processor. Thus, by counting the instructions in the program code, it is possible to give the number of cycles needed for the execution of the program. However, the time needed for I/O instructions depends on the load of the processor buses used by all processor units. Due to the hardware-supported multithreading, threads waiting for I/O instructions can swap out, leaving the processor to threads that are ready to run, thus keeping the processor at full load. For this performance evaluation it is assumed that I/O operations always terminate fast enough for keeping the processor busy at all times.

The results of the these calculations are shown in Fig. 8 for different configurations of the BAM. The performance is given in packets per seconds for different timer granularities, which result from multiplications of the clock period by a power of two and are given as rounded values in the chart. The results for 27.3 μs and 13.7 μs granularity drop to zero as soon as the timer condition (see 4.2) cannot be fulfilled any more, since giving the performance for configurations at which proper operation of the timer mechanism cannot be guaranteed does not make sense.

As shown, performance decreases approximately linear for an increasing number of queues, since every additional queue needs additional processing time for a timer and for finalization. However, the curves are not straight due to effects like processing deactivated timers and writing parts of the header for every fourth packet.

In order to give an impression of the performance that can be achieved by the BAM, the maximum packet rate that can occur in the NP using our hardware is shown as a horizontal line in the chart. With the three Gigabit Ethernet ports, a maximum packet rate of 4.5 Million packets per second is possible considering the smallest Ethernet frame size. As shown in the chart, the BAM can handle 64 queues with a timer granularity of 54.6 μs per timer at this rate.

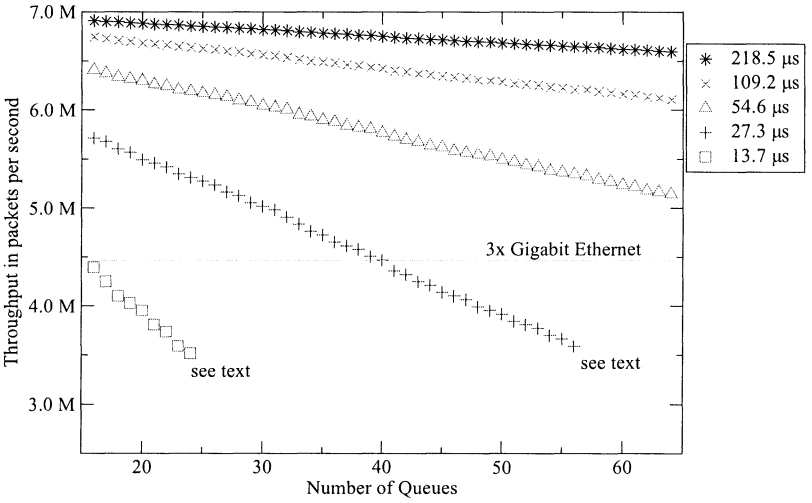

Figure 8. Calculated throughput of the BAM for different timer granularities

7. Conclusion

In this work, a Burst Assembly Unit on a Network Processor has been implemented and evaluated. It was shown that a Network Processor is suitable for such an implementation although the characteristics differ from typical network applications. Both assembly strategies – time-based and size-based – have been realized. For this, an efficient timer mechanism has been found that is able to provide Burst Assembly at line speed.

In a detailed performance evaluation, a method to calculate the maximum packet rate for a certain setup has been developed. With this, the throughput for a number of relevant scenarios has been calculated. It was shown that the Burst Assembly Module can handle up to 6.8 Million packets per second.

As a next step, the task of Burst Disassembly, which is currently implemented on a PC for testing purposes, has to be implemented as a software module for the NP. This module then could be integrated in the NP that also runs the BAU, in order to provide complete edge node functionality on an NP.

References

Dolzer, K.: Assured Horizon - A New Combined Framework for Burst Assembly and Reservation in Optical Burst Switched Networks, Proceedings of the European Conference on Networks and Optical Communications (NOC 2002), Darmstadt, 2002.

Gauger, C. M.: Trends in Optical Burst Switching. Proceedings of SPIE ITCOM 2003, Orlando, 2003.

Intel Corporation: Intel Internet Exchange Architecture Portability Framework Developer's Manual, 2003

Iperf bandwidth measurement tool: http://dast.nlanr.net/Projects/Iperf

Qiao, C., and M. Yoo: Optical Burst Switching - A New Paradigm for an Optical Internet, Journal of High Speed Networks, Special Issue on Optical Networks, Vol. 8, No. 1, pp.69-84, 1999.

Vokkarane, V.M. ,K. Haridoss, and J. P. Jue: Threshold-Based Burst Assembly Policies for QoS Support in Optical Burst-Switched Networks. Proceedings, SPIE Optical Networking and Communication Conference (OptiComm) 2002, Boston, MA, pp.125-136, 2002.