Programmable Subsystems: A Traffic Shaping & Packet Prioritizing Case Study

P. Halvorsen[1,2], Ø.Y. Sunde[1], A. Petlund[1] and C. Griwodz[1,2]

[1]Department of Informatics, University of Oslo, Norway [2]Simula Research Lab., Norway
e-mail: {paalh, oysteisu, andreape, griff}@ifi.uio.no

Abstract

Due to a faster speed increase of networks than processors, we have today a trend towards the distribution of functionality and workload for network processing on several processing units. For this purpose, network processors are being developed which are special processor architectures aimed for demanding networking tasks such as backbone routing and switching. In this paper, we investigate the possibility of improving the scalability of intermediate nodes by offloading the packet processing workload on the host, and in particular, we present a traffic shaper and packet prioritizer implemented on the Intel IXP2400.

Keywords

IXP2400, workload distribution, intermediate nodes

1 Introduction

Since its beginning as an experimental network, the Internet has had a huge development. Especially in the last ten or fifteen years, it has become a common property, and a variety of services is now available to the public. With the emergence of modern applications, like high rate, time dependent media-on-demand and latency restricted online games, we must support a large variety of traffic classes and face an increasing need for packet processing power. However, we now see that CPU speed development does not keep pace with Moore's law and the network bandwidth increases faster (Comer, 2004), i.e., the data rates achievable on modern network links are larger than what a state of the art processor can manage.

Network processing units (NPUs) are special processor architectures aimed for demanding networking tasks such as backbone routing and switching. A typical NPU has several small, symmetric processing units working concurrently. The parallel structure of the packet processing enables the tasks to be performed in a pipeline, with each functional unit performing a special task. In this case, since the communication services can be offloaded to a separate device, the bottleneck of single CPU processing where processing capacity has to be scheduled between various tasks on the host can be removed with respect to packet processing.

Companies such as Agere, IBM and Intel are manufacturing NPUs for different platforms and purposes. All these are more or less based on the same offloading ideas, but their implementations vary greatly. As the NPUs come with a large amount of resources, we are currently investigating how to improve the scalability of intermediate nodes in which the NPU offloads the host with operations ranging from low level packet processing to application specific processing (Hvamstad et al., 2005). In this paper, we evaluate an NPU based on Intel IXP2400 in general, i.e., with respect to its capability to offload a host machine, and we describe a case

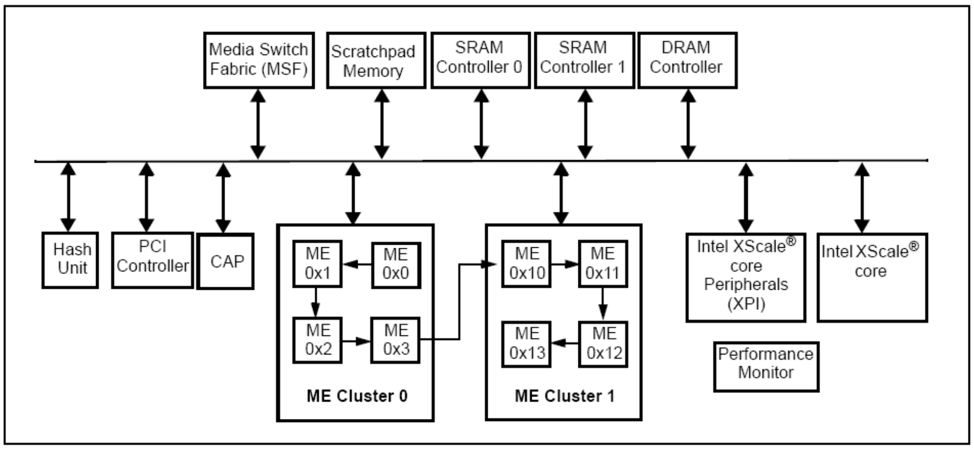


Figure 1: IXP2400 block diagram (Intel Corporation, 2004)

study of how traffic shaping and packet prioritizing can be implemented using NPUs. The experiments using our prototype show that the IXP2400 can offload networking tasks from the host and that shaping and prioritizing can be efficiently performed on the network card.

The rest of this paper is organized as follows. Section 2 gives a small overview of examples on related work with respect to network processors, and in section 3, we describe and evaluate the Radisys ENP2611 IXP2400 NPU. Then, section 4 presents and gives an evaluation of our implementation of the traffic shaper and packet prioritizer. Finally, we give a conclusion and directions for future work in section 5.

2 Related Work

On-board network processing units have existed for some time with the initial goal of moving the networking operations that account for most CPU time from software to hardware. Most existing work on network processors concentrates on traditional networking operations like routing (Kalin and Peterson, 2001; Spalink et al., 2001) and active networking (Kind et al., 2003), while only a few approaches have been proposed to additionally perform application specific operations for which the modern NPUs have resources. One example is the booster boxes from IBM (Bauer et al., 2002) which try to give network support to massive multi-player on-line games by combining high-level game specific logic and low-level network awareness in a single network-based computation platform. Another example is the NPU-based media scheduler (Krishnamurthy et al., 2003) where a media stream can go through the system without involving the host CPU.

Thus, distribution of functionality to NPUs is becoming more common, but the few existing examples have used old hardware. However, as the new generation of NPUs like the IXP2400 have more resources, one can offload more functionality to the network processor. In our work, we therefore look at how to scale intermediate nodes better using NPUs where the NPU additionally perform application specific operations.

3 Intel IXP2400

The IXP2400 chipset (Intel Corporation, 2004) is a second generation, highly programmable NPU and is designed to handle a wide range of access, edge and core applications. The major

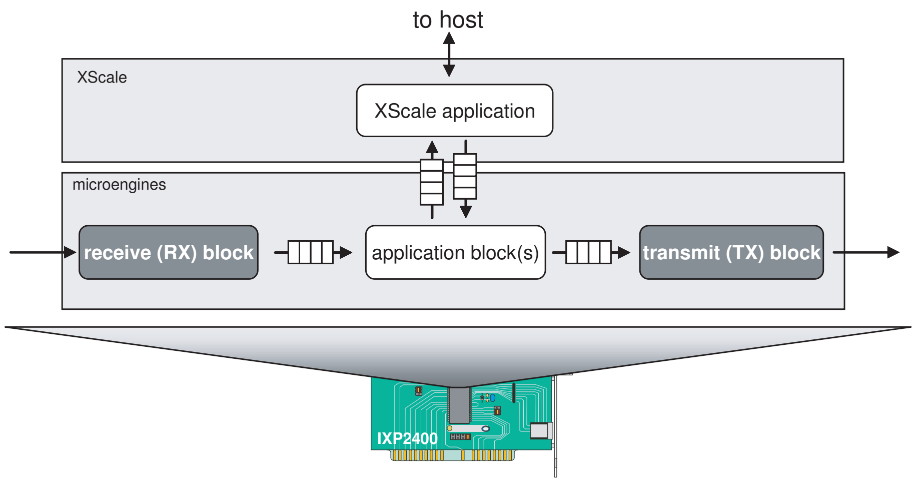


Figure 2: Block layout

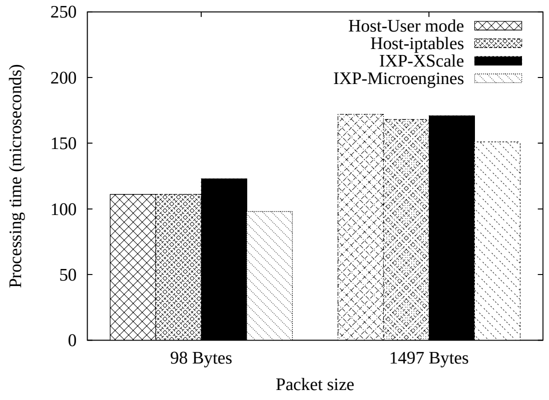
functional blocks of the IXP2400 chipset are shown in figure 1. The basic features include an 600 MHz XScale core running Linux, eight 600 MHz special packet processors called microengines (μEngines), several types of memory and different controllers and busses. With respect to the different CPUs, the XScale is typically used for the control plane (slow path) while μEngines perform general packet processing in the data plane (fast path). The three most important memory types are used for different purposes according to access time and bus bandwidth, i.e., 256 MB SDRAM for packet store, 8 MB SRAM for meta-data, tables and stack manipulation, and 16 MB scratchpad (on-chip) for synchronization and inter-process communication among different processing units. Moreover, the physical interfaces are customizable and can be chosen by the manufacturer of the device on which the IXP chipset is integrated. The number of network ports and the network port type are also customizable.

Figure 2 shows a typical application setup. The gray boxes are components included in the SDK and the application programmer can do whatever he wants in between. The receive (RX) microblock receives a data packet from a ports, and passes a reference to the data packet on a scratch ring[1]. An application on μEngines or XScale can dequeue the reference, find and process the data, and when the packet is ready for transmission, a buffer handle is put on a scratch ring read by the transmit (TX) block.

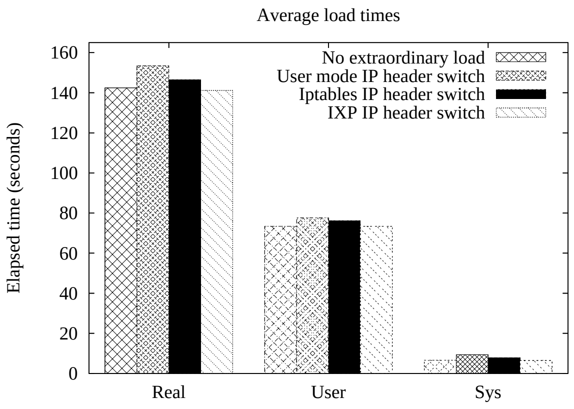
3.1 Experiments

To test the general performance of the IXP2400, we have performed several tests on a Radisys ENP2611 network card, which uses the IXP2400 chipset. Our first test is an IP header switch application to see the forwarding latency of the NPU compared to a Pentium4 3 GHz box[2] running Suse Linux 9.3 (kernel 2.6.11). Figure 3(a) shows the round trip times of packets in a point-to-point network using 1) the IXP μEngines only, 2) the IXP μEngines and XScale, 3) the host kernel and 4) a user-space process on the host. Data was collected using *tcpdump*. The results show that, when using the low-level μEngines, the system has a 11-12% improvement, which means that the IXP2400 is an efficient means to distribute functionality and perform packet processing operations at a low level.

[1]A scratch ring is a hardware assisted ring with atomic operations that is used for interprocess communication.
[2]Note that the host CPU has *five* times the clock frequency compared to the NPU.



(a) Average time for IP header switching



(b) Time spent on a `tar` process.

Figure 3: General offloading effects using an NPU

Furthermore, to show the effect of offloading, we performed a CPU intensive task (`tar -z`, i.e., tar and gzip, on an 870 MB file) on the host concurrently with the IP header switching application running on either the host or the NPU. Figure 3(b) shows the average time to perform the `tar` operation with respect to the used data path. As one can see, the consumed time is reduced (by approximately 8-9%) when our low rate network traffic is offloaded to the IXP card, and the packet forwarding time also similarly increases when performing the networking operations on the host.

3.2 Discussion

Offloading network functionality to μEngines will, in other words, not only free host resources, but also improve the processing and forwarding speed compared to what the host would be able to handle. Additionally, the offloading effect shown above is marginal as the IP header switch application is far less resource demanding compared to several existing Internet applications today like HTTP video streaming. Thus, with the emergence of high rate, low latency applications the importance of workload distribution will increase.

4 Traffic Shaping and Packet Prioritizing

In the Internet today, several applications require some notion of quality of service (QoS) support, and it is important to optimize or guarantee performance. To deal with concepts like classification, queue disciplines, enforcing policies, congestion management, and fairness, both traffic shaping and packet prioritizing are important means. However, while giving better control of network resources, these operations impose a higher load on the intermediate nodes with respect to processing. We have therefore implemented a prototype of these mechanisms using the IXP2400 NPU.

Figure 4 shows an overview of the implemented prototype which for now is targeted for a video streaming scenario. The shaper and prioritizer is meant to be deployed on an intermediate node at the ingress to a network with flow control or reserved resources. It is meant to be used with a layered video codec where each layer has a separate priority. On the XScale, we have a small component initializing the system and loading the microblocks onto the microengines. All the real work is performed by the μEngines where we have pipelined the functionality using three

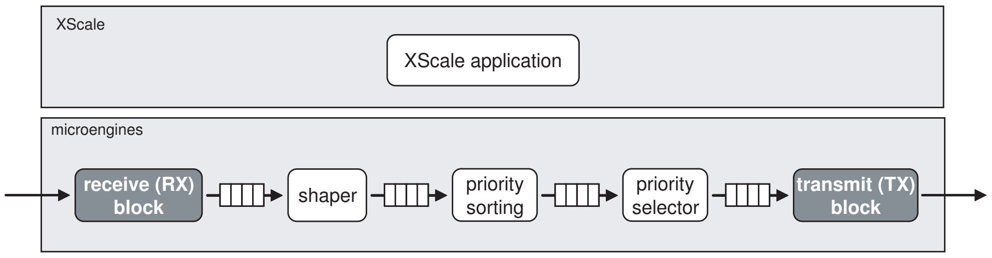


Figure 4: Prototype blocks for shaping and prioritizing

different microblocks which can use one or more μEngines according to the resource demands of a task.

4.1 The Shaping Microblock

To be able to control the volume of traffic being forwarded by and sent from our intermediate node to the network (bandwidth throttling and rate limiting), a microblock running on the μEngines debursts the traffic flows, i.e., smoothes the peaks of data transmission.

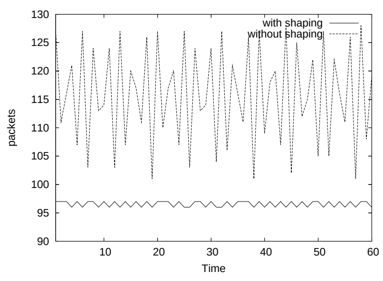
The implementation is based on token buckets, where each packet stream (initialized using the RTSP DESCRIBE and SETUP messages and the servers' responses to these) is given a token rate and bucket depth according to the requested or negotiated rate. With respect to packet dropping in case of bucket underflow (exceeding the assigned rate), we have implemented a queue head-drop, i.e., packets with the shortest or expired deadline will be dropped.

To test the shaper we made some basic experiments by streaming data with and without using the traffic shaper. The stream had an average reservation of 800 Kbps using the shaper, and the sender sent 1024 bytes packets. For different tests, we used different sending rate fluctuations (switching rate for each 100th packet). Figure 5 shows the results of three different measurements. Figure 5(a) shows a scenario where the sending bandwidth fluctuates between 1100 Kbps and 800 Kbps. As we can see, the uncontrolled stream takes more resources than allowed, and the rate oscillates with large peaks. The shaped stream, however, has a smooth output stream slightly below the 800 Kbps limit and drops packets according to the implemented dropping scheme. Reducing the lower sending rate (larger fluctuations) makes the average closer to the allowed rate, but the peaks without a shaper are higher as the plot in figure 5(b) indicates. Finally, in figure 5(c), the lower rate is dropped to 600 Mbps, and in this case, we also have some fluctuations in the shaped output stream due to a build-up of available tokens. In conclusion, our prototype on the NPU gives a shaped stream, and since it is independent of the host load, the experienced latency has much smaller variations.

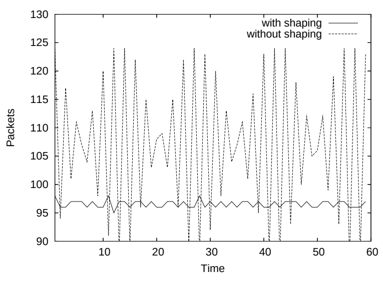
4.2 The Prioritizer and Packet Scheduler Microblocks

A priority-based scheduling of packets is necessary to be able to provide different service classes and some notion of guarantees. To support this, we have implemented a packet prioritizer on our NPU. For now, our system handles four different priorities, and in our video streaming context, the priorities are set in the RTP headers.

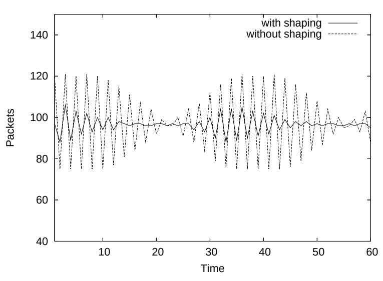
In the current implementation of the prototype, the functionality is distributed on two μEngines. The first sorts packets according to the priority and places them onto the respective priority queue. The queues are implemented using the hardware supported scratch rings, so the sorted



(a) 1100 and 800 Kbps

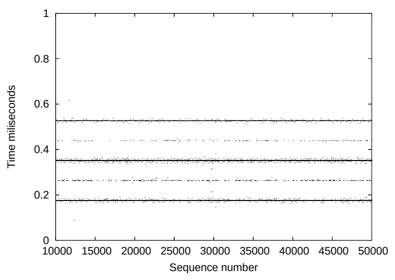


(b) 1100 and 700 Kbps

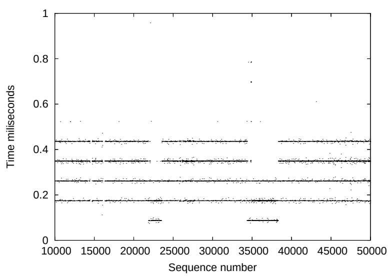


(c) 1100 and 600 Kbps

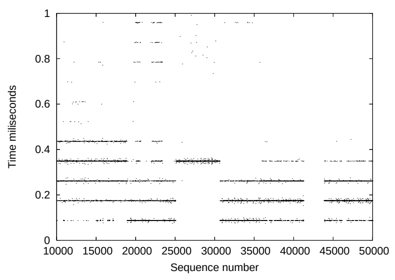
Figure 5: Packet rate with and without the traffic shaper using different rate fluctuations



(a) IXP



(b) Host, no load



(c) Host, disk I/O & zip load

Figure 6: Time gap between first priority packets (100 Mbps video stream)

packets are directly available to the next μEngine running the priority scheduler. The scheduler then transmits packets based on the priorities and the given resource assignment, and in the case where a queue is empty or does not utilize its reserved resources, the scheduler is work-conserving and proceeds with the next round at an earlier time.

In order to test the prioritizer, we have performed several simple tests on an intermediate node using a prioritizer either on the NPU or in the Linux kernel on the host (using `tc qdisc`), i.e., no fairness. The measurements are performed on an 80 Mbps (layered) video stream. Additionally, we had a background traffic load of 80 Mbps and had several workloads appropriate for a multimedia proxy, i.e., reading and writing data from disk (file copy operations) and processing data using the CPU (`gzip`). Our NPU implementation is unaffected by host workload, but the host mechanisms battle with the rest of the system for resources and the jitter is thus greatly affected.

The effects of offloading the prioritizer to an NPU are shown in figures 6[3], 7 and table 1. Figure 6 plots an arbitrary window of the 150.000 first priority packets sent, figure 7 and table 1 show the average time gap between packets and the respective variance. While the jitter (measured as the time gap between packets) of the first priority packets using the NPU-based prioritizer is usually small (see figure 6(a)), the host implementations show much more jitter. Figure 6(b) shows the system with no extra workload where the jitter and the variation are larger than on the NPU. When we additionally add a light web-cache or VoD like workload on the host the problems further increase (see figure 6(c)). In conclusion, the NPU implementation has by far the lowest variance (shown by the standard deviation in table 1 and the plots), and it is independent of the varying host workload.

[3]The clear horizontal lines in the figure are due to the clock granularity. The other (lower) priority packets have similar results compared to best-effort traffic.

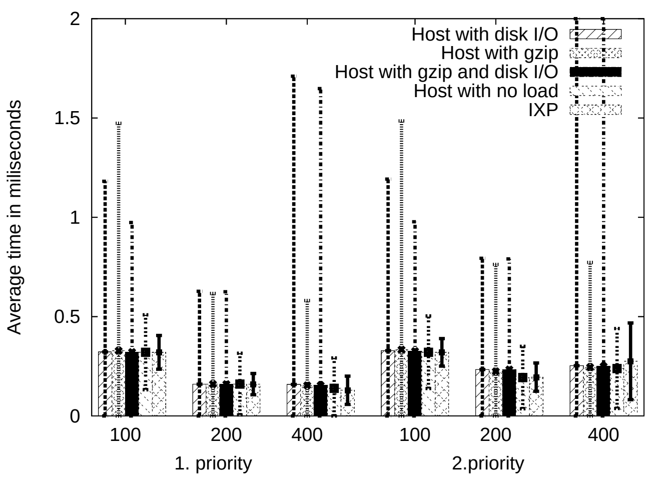


Figure 7: Average time gap with error bars

	IXP	host				IXP	host			
load		no	disk I/O	zip	disk & zip		no	disk I/O	zip	disk & zip
maximum	0.856	43.511	200.008	364.494	123.908	20.390	56.459	829.084	49.370	662.014
minimum	0.000	0.000	0.000	0.004	0.007	0.000	0.003	0.000	0.003	0.003
average	0.320	0.321	0.322	0.328	0.321	0.129	0.139	0.159	0.154	0.154
median	0.351	0.349	0.348	0.348	0.348	0.088	0.089	0.174	0.174	0.174
stdev	0.085	0.189	0.859	1.145	0.652	0.071	0.153	1.552	0.427	1.494
	100 Mbps video stream					**400 Mbps video stream**				

Table 1: Time gap statistics between first priority packets (two selected tests)

4.3 Discussion

As we can see from the tests, the NPU has no problems of performing the shaping and prioritizing in an efficient manner. The advantage of using the NPU to perform these operations is that the application specific evaluations of each packet can be executed in a network-aware environment without the need to move the packet up to the application itself running in user space on the host. Thus, we save a lot of data transfers, interrupts and context switches which greatly influence the performance and the experienced latencies. Our prototype shows that we can achieve a shaped, prioritized stream independent of the host load, i.e., less variations in latency. This is for example beneficial in the context of proxy caches both forwarding data from the origin server and servicing clients from the cache's local storage on the host.

However, with respect to the implementation, there are several future issues. For example, the operation of dropping packets in the shaper should be performed with respect to both deadlines and priorities, e.g., dropping a base layer packet in a layered video stream implicitly invalidates the corresponding enhancement packets. Additionally, the number of priority levels is now restricted to the number of available rings. In total, there are 16 scratch rings, but these are shared for the whole system. Our current prototype uses nine of the available scratch rings. To support more priority levels than we have scratch rings available, a different queuing mechanism must be implemented. Furthermore, yet another task is to make further experiments to see how to best utilize the resources on-board (we still have not fully utilized all resources) and efficiently pipeline the functionality.

5 Conclusion and Future Work

In this paper, we have investigated the possibility of using an NPU to improve scalability of intermediate nodes. In general, the evaluation of our prototype shows that offloading network functionality to μEngines will both free host resources and improve the processing and forwarding speed compared to what the host would be able to handle. In our case study prototype performing traffic shaping and packet prioritizing, our tests show that the NPU can efficiently perform such operations on behalf of the host and that we still have a lot of resources to perform more (application level) operations.

With respect to ongoing work, we are currently performing more extensive tests and evaluations, and we are working on several components in the context of servers and intermediate nodes. These include caching functionality support for proxy caches and a cube architecture for a media server - both being currently implemented on the IXP2400. Finally, we aim at integrating all of our components in order to make a scalable, high-performance intermediate node.

References

Bauer, D., Rooney, S. and Scotton, P. (2002), Network infrastructure for massively distributed games, *in* 'Proceedings of the Workshop on Network and System Support for Games (NETGAMES)', pp. 36–43.

Comer, D. E. (2004), *Network Systems Design using Network Processors - Intel IXP version*, Prentice-Hall.

Hvamstad, Ø., Griwodz, C. and Halvorsen, P. (2005), Offloading multimedia proxies using network processors, *in* 'Proceedings of the International Network Conference (INC)', pp. 113–120.

Kalin, S. and Peterson, L. (2001), Vera: An extensible router architechture, *in* 'Proceedings of the IEEE Conference on Open Architectures and Network Programming (OPENARCH)', pp. 3–14.

Kind, A., Pletka, R. and Waldvogel, M. (2003), The role of network processors in active networks, *in* 'Proceedings of the IFIP International Workshop on Active Networks (IWAN)', pp. 18–29.

Krishnamurthy, R., Schwan, K., West, R. and Rosu, M.-C. (2003), 'On network coprocessors for scalable, predictable media services', *IEEE Transactions on Paralell and Distributed Systems* , Vol. 14, No. 7, pp. 655–670.

Intel Corporation (2004), 'Intel IXP2400 network processor datasheet'. URL: ftp://download.intel.com/design/-network/datashts/30116411.pdf

Spalink, T., Kalin, S., Peterson, L. and Gottlieb, Y. (2001), Building a robust software-based router using network processors, *in* 'Proceedings of the ACM Symposium of Operating Systems Principles (SOSP)', pp. 216–229.