A mapping strategy for resource-efficient network processing on multiprocessor SoCs*

Matthias Grünewald Jörg-Christian Niemann Mario Porrmann Ulrich Rückert

Heinz Nixdorf Institute and Department of Electrical Engineering,
University of Paderborn, Germany
{gruenewa,niemann,porrmann,rueckert}@hni.upb.de

Abstract

Hardware architectures based on a field of hardware-extended processors can provide flexible computing power for applications where parallelism can be exploited. For multiprocessors, the assignment of functionality to execution units can have a great impact on the performance. Additionally, finding the optimal mapping can be a time-consuming task. We present a multiprocessor architecture along with a suitable design method that includes an automated solution to the mapping problem. Our hardware architecture employs a network-on-chip (NoC) to achieve a high degree of scalability for the application and for the system in respect to future integration technologies. We also show how to reduce the packet buffer requirements with a proper scheduling strategy and present first estimates for the resource consumption of an application targeted for mobile networking.

1. Introduction

Networking is an ideal application for multiprocessors when several packet flows have to be processed in parallel. The need to quickly adapt to changing network protocols has led to the development of *network processors* (NPUs) that contain a pool of programmable forwarding engines along with specialized hardware assists. A great challenge for implementing network protocols on NPUs is to decide which component executes which part of the protocol. The designer has to maximize the performance, e.g., the throughput per packet, while meeting constraints such as memory restrictions and communication bandwidth between the functional units. The design space increases even more if the hardware architecture of the NPU has to be changed to meet the design goals. Parameters such as memory size, processor frequencies and link data rates are variable and the best setting has to be found. We are developing a framework for designing and implementing network protocols with resource-efficient, parallel System-on-Chips (SoCs). In contrast to most existing work about NPUs, we also consider the energy consumption of the system. Within the framework, the hardware architect is offered a method to estimate the effect of different parameter settings on the resource consumption, and the software designer can employ an automated way to map the protocol functions to the units of the system.

There exist only a few comparable approaches. Franklin et al. [3] present an analytical model for an NPU that consists of a number of processor clusters where each cluster has its own external memory. The received packet flows are uniformly distributed by a packet demultiplexer. Our approach allows a more fine-grained distribution of protocol parts to processors that are equipped with smaller, but faster and less energy-consuming on-chip memory. Thiele et al. [3] describe packet processing with a task graph and use a system model based on network calculus to obtain the delay and buffer usage characteristics of the application. The mapping of the task graph to the system components is found by using evolutionary search techniques. Their approach has a high degree of generality and can be applied to many different heterogeneous system architectures. Our approach is predominantly tailored for the design of a flexible hardware architecture where the trade-offs between different system parameters need to be analyzed.

We are applying a modeling technique that is similar to the one by Thiele et al. It allows a parallelizable description of the protocol processing. The software implementation of the network protocol is done within a hardware-independent software library and tested inside our network simulator SAHNE [10]. The execution environment for the software implementation is provided by a multiprocessor hardware architecture. It uses a network-on-chip to solve many deep-submicron design problems such as clock distri-

* This work was supported in part by the DFG-Sonderforschungsbereich 376 “Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen” and Infineon Technologies AG, especially the department CPR ST, Prof. Ramacher.

1530-1591/04 $20.00 (c) 2004 IEEE

bution, structuring and reuse of global wires, and scalability. Flexibility is obtained by using a general-purpose processor within the processing engines. Resource-efficiency is gained by analyzing the impact of additional hardware extensions in the engines on the resource consumption. The resource consumption of the protocol processing can be characterized by running the protocol on a single simulated or emulated processing engine that has special profiling support. Our methodology is perfected by a method that finds an energy- or delay-optimized mapping of the protocol implementation to the processing engines of the system. It also allows an exploration of important system parameters such as the number of processing engines and link data rates of the NoC and computes their effect on throughput, delay, buffer requirements, energy and area.

There already exists a large variety of hardware architectures for NoCs. They differ in the applied connection topology (folded torus [4], fat-tree [6], grid [9, 2]), area requirements and the communication services provided. However, none of these approaches have yet classified the influence of the specific communication characteristic of the application on the performance of the NoC. In this paper we show how to close this gap. We focus on the applied scheduling and mapping strategies. Section 2 gives an overview about the packet processing and system model we have applied. In section 3, we describe the scheduling algorithm used for processing packet flows and exchanging them between processing engines. The mapping approach is presented in section 4, and section 5 uses an example of how the performance of a medium access protocol for mobile ad hoc networks can be tuned.

2. Modelling packet processing architectures

2.1. Flow processing graph

We describe the functionality that the SoC has to provide by a *flow processing graph* $FPG = (M, S)$. The nodes $m \in M$ represent *packet methods*. The edges $s \in S$ represent *flow segments*. The packet methods generate and process the packets, and the flow segments describe how the packets are forwarded between the methods. An instance l of a *protocol layer* consists of a set of methods $M_l \subset M$ that share a common state Z. A *packet flow* $pf \in PF \subset S^*$ is a tuple (list) of flow segments that describes the way of a packet flow through the graph. We are currently using three packet methods per layer. The two methods ipp and opp have to be employed when packets from an inbound flow or outbound flow have been received. The third function bpp is called at user-defined time points. For example, it can be used for implementing flow-independent maintenance tasks such as the routing table cleanup. The calling period of the bpp method is specified by using virtual flow segments. There also exist special layers lp called *logical ports* that denote the source and sinks of the packet flows that are processed by the system. The set L_{lp} contains all logical ports. The flow processing graph has a special structure that is exploited by our mapping algorithm. It consists of local and global layer instances (cf. fig. 1). The local layers are connected to logical ports in a pipelined fashion. The user defines for every logical port lp a set of local layers $L_{lp,local}$ that contains the layers that work for the port. The global layers are connected to several local layers. The set L_{global} contains the global layers of the graph.

Figure 1 shows an exemplary flow processing graph. It describes a novel medium access protocol for mobile ad hoc networks (MANETs) [10]. Each node of this MANET can submit in eight directions, dividing the space in eight sectors. The flow processing graph consists of one local layer for each sector (*Sector-MAC*) and a global layer (*Neighbor-MAC*) that maintains the list of neighbors. In each sector, the network node maintains a link only to the neighbor that requires the least transmission power. In a periodically occurring time window, other nodes can send requests to change the link if they require less transmission power. The local and global layers synchronize themselves if network changes occur. Data packets are exchanged between the networking (*NET*), MAC and physical (*PHY*) layers via the local MAC layer instances. The use of a contact time window has the advantage that the bandwidth of the flow segments between the local and global layers can be kept low. Figure 1 does also show an exemplary packet flow. It is the usual path a MAC control packet takes when it is received from a neighbor. The global MAC layer generates an answer that is submitted back to the neighbor. The local MAC layer uses a time-triggered bpp method to submit the answer back within the contact window of the neighbor.

We have implemented this protocol within an ANSI-C-based software framework called *packet processing library* (PPL). It has low memory requirements, e.g. the size of the program code and data for the example application, compiled for the Motorola M*Core compatible S-Core used in our hardware architecture, is below 70 kB. By using our profiling environment PERFMON [5] and adding profiling support for flow segments to the PPL, we have measured the resource consumption of packet methods and flow segments. The annotations in the flow processing graph (cf. fig. 1) show the results. The numbers in the methods denote their maximum execution time ET_m in cycles. The numbers close to the flow segments denote their bandwidth scaling factor, minimum packet size $p_{s,min}$ and maximum packet size $p_{s,max}$. The bandwidth requirement B_s of flow segment s is obtained by multiplying the scaling factor with the port data rate B_{lp}. For virtual flow segments, the calling period of the method is annotated.

2.2. SoC architecture

An instance of a packet processing system is described by a graph $SYS = (G, H)$ whose nodes $g \in G$ represents

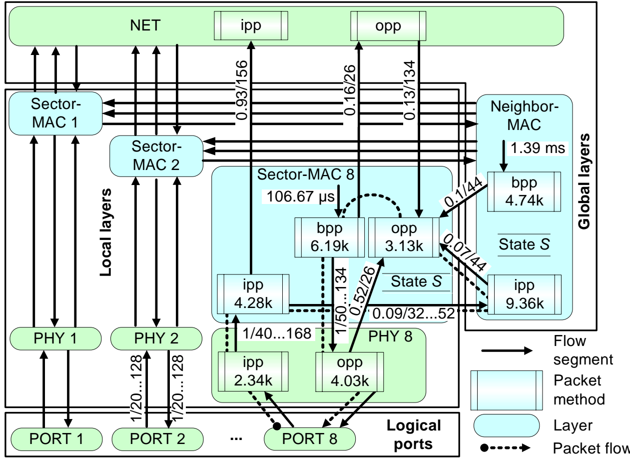

Figure 1: Annotated flow processing graph

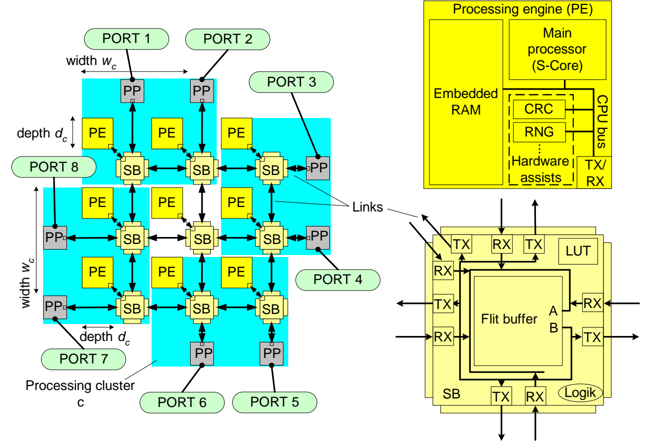

Figure 2: Processing engine, switch box and a system

the used *modules* and whose edges $h \in H$ represent the communication *links* between the modules (cf. fig. 2). The modules can be divided into three sets $G_{PE} \cup G_{SB} \cup G_{PP} = G$ where G_{PE} describes the set of *processing engines* (PEs) and G_{SB} the set of *switch boxes* (SBs) that form the NoC. G_{PP} describes the set of *physical ports* that denote the physical connections used to inject and receive external packet flows. Each border PE has only one nearest physical port to ensure a uniform load at the border. PEs contain a main processor, embedded memory, hardware assists and a *link interface* (transceiver) to connect the SB. PEs can only use the local memory, no access to external memory is possible. Our SB variant contains a dual port shared memory for storing incoming data, a look-up table (LUT) with control logic for scheduling the forwarding of flow segments and a transceiver (TX/RX) for each link. The association of logical ports to physical ports and the partitioning of the system in processing clusters c is explained in 4.

3. Scheduling

For processing the flow graph on the system, the execution of packet methods on PEs and the forwarding of flow segments between PEs have to be scheduled. We use *generalized processor sharing* (GPS) [7] for this purpose. For every flow segment s, a FIFO buffer and a weight ϕ_s is allocated. The algorithm guarantees that the packets of the flow segments are processed in parallel at an individual service rate of $\phi_s / (\sum_{\tilde{s} \in S(t)} \phi_{\tilde{s}}) B_{cm}$ where B_{cm} is the total service rate of the system component cm. The set $S_{cm}(t)$ contains the backlogged segments at time t. The worst case delay a flow segment experiences depends on the utilization U_{cm} of the system component [7]:

$$D(s) \leq \frac{p_s}{\phi_s \cdot B_{cm}} \cdot \sum_{\tilde{s} \in S_{cm}} \phi_{\tilde{s}} = \frac{p_s}{\phi_s \cdot B_{cm}} \cdot U_{cm} \quad (1)$$

where p_s is the size of the packet in bits.

For every flow segment whose layers are assigned to different modules, a path of links has to be set up through the NoC that supports the bandwidth and delay requirements. We use a practical implementation of GPS called *packetized general processor sharing* (PGPS) [7] to enable a non-blocking flow segment forwarding on links. PGPS implements GPS by defining finish times for each packet that has been received from the different flow segments. The packet with the earliest finish time is forwarded next at the full data rate B_h of the link h. The finish time of PGPS is at most the finish time of GPS plus the time needed to process the largest packet at the maximum service rate [7]. This is due to the fact that the processing of the current packet can not be suspended anymore if a new packet arrives that has an earlier finish time. However, if all packets have the same length and are received on synchronized time points, the current packet will be finished when the next packets arrives (if the input data rate is equal to the output data rate). Hence, the proof can be easily extended to show that the finish times of PGPS are equal to GPS for this case. Therefore, we divide the packets of flow segments in data units called *flits*. Each flit has a fixed size of q bits and contains a header of size $\hat{q}$ to denote the flow segment it belongs to. We choose the weight $\phi_{s,h}$ per link such that the forwarding time is below the reception time of the next flit. For this purpose, the utilization of the link U_h has to be below one:

$$\phi_{s,h} = \frac{1}{\lfloor B_h (1 - \frac{\hat{q}_h}{q_h}) / B_s \rfloor}, \quad U_h = \sum_{\tilde{s} \in S_h} \phi_{\tilde{s},h} \leq 1 \quad (2)$$

where S_h are the flow segments that are forwarded on link h. To obtain the delay per flit, p_s in eq. 1 has to be replaced by q. The advantage of this virtual circuit switching is that each SB has to hold, at most, two flits ($2q$) per flow segment in its buffer. Since the flits can have a length that is

only a fraction of the maximum packet length, the buffer usage Q_{SB} in the SB can be kept low. This tackles one important problem of NoC design since on-chip memory has high area requirements. Compared to the approaches in [4, 9], GPS scheduling does also not reserve link bandwidth. Hence, other flow segments are forwarded at a reduced delay if one flow segment is not active.

For applying GPS inside the PEs, the service rate B_{cm} in eq. 1 has to be replaced by the frequency f_{PE} of the PE. Additionally, the packet length p_s has to be substituted by the execution time $ET_{m,PE}$ of the packet method m on the PE that processes s. We propose to use a real-time scheduling algorithm such as earliest deadline first to implement GPS. It can suspend the execution of the current packet method if a packet arrives that has an earlier finish time. The weight $\phi_{s,PE}$ has to be set in such a way that the processing time is below the reception time of the next shortest packet. This can only be ensured if the link utilization U_{PE} is below one. The processing time depends on the frequency f_{PE} of the PE and the execution cycles $ET_{m,PE}$ of the packet method. It can vary from one PE to another due to different hardware assists. The weight and utilization are obtained by

$$\phi_{s,PE} = \frac{ET_{m,PE}}{\lfloor p_{s,min} \cdot f_{PE}/B_s \rfloor}, \; U_{PE} = \sum_{s \in S_{PE}} \phi_{s,PE} \leq 1 \quad (3)$$

where $p_{s,min}$ is the size of the shortest packet of s in bits and S_{PE} is the set of flow segments that are processed by the PE. This also limits the buffer space inside the PE to two packets per flow segment. If a flow segment is generated and processed at different PEs, the link interface of the source PE has to convert the packets into flits and vice versa. Therefore, these flows require an additional output buffer space of two packets at the source PE that is used by the link interface. As a packet can consist of several flits, the link interface of the PE must provide a traffic shaping functionality. Otherwise, the PGPS scheduler would transmit all flits immediately to the SB if the uplink is free and the buffer constraint of two flits per flow segment could not be held anymore. Please note that eq. 3 does not consider the overhead for task switching.

4. Mapping the application to the system

The goal is to find a mapping of layers to PEs that minimize a given cost function while meeting the technology constraints. We define this problem as an integer linear program (ILP), inspired by Prakash et al's approach [8]. Additionally, we use a hierarchical approach to keep the number of variables small and, with this, the solution time short. Our approach utilizes the partitioning of the flow processing graph in global and local layer instances. To exploit the concept of locality, our approach ensures that local layers are processed close to the physical port to which the logical port will be assigned to. The whole mapping algorithm is outlined in fig. 3.

```
map (FPG, SYS)
begin
  clear all variables and sets;
  partition SYS into local processing clusters;
  assign uniformly logical ports to physical ports;
  for all local clusters c do
    for all PP ∈ G_{PP,c} do
      for all lp ∈ L_{lp,PP} do
        assign all local layers to cluster c: L_c := L_c ∪ L_{lp,local};
      end
    end
    {L_PE | PE ∈ G_{PE,c}} := solve local ILP to map L_c to G_{PE,c};
  end
  assign all layers and PEs to a new global cluster ĉ;
  {L_PE | PE ∈ G_{PE,ĉ}} := solve global ILP to map L_global to G_{PE,ĉ};
  create a sorted list T of flow segments s ∈ S, highest B_s first;
  for all s ∈ T do
    disable for this iteration links that do not have left enough band-
    width (U_h + φ_{s,h} > 1) or buffer space (Q_SB + 2q > Q_{SB,max});
    H_s := compute minimal {energy, delay} path between source and
    destination PEs of s;
    update utilization U_h of all links and buffer space Q_SB of all SBs;
  end
end
```

Figure 3: The mapping algorithm

To map the application to the system, the border of the system is first divided into processing clusters c that have a width of w_c and a depth of d_c PEs (cf. fig. 2). For every cluster, a set of PEs $G_{PE,c}$, a set of SBs $G_{SB,c}$ and a set of physical ports $G_{PP,c}$ is obtained. For each physical port PP, a set of assigned logical ports $L_{lp,PP}$ is created in such a way that the logical layers are distributed uniformly (cf. fig. 2). All local layers that work for the a logical port are assigned to the processing cluster the physical port belongs to. Now, for every cluster, the layers that belong to the cluster are assigned to the PEs in the cluster by solving the ILP. After each mapping step, a set of layers L_{PE} is obtained for each PE that contains the layers that are executed on the PE. The global layers are mapped after all local processing clusters have been processed.

The ILP uses a binary variable $b_{l,g}$ to decide if the layer l is assigned to module g. Due to the iterative cluster mapping, there are some special cases that result in a fixed value of the binary variable. These are: a layer has already a fixed assignment to a module (e.g. for already mapped local layers), a layer lies outside the current cluster c or a module is a physical port (no layers can be executed on the port). By taking these special cases into account, the contents of the binary variable can be described as:

$$b_{l,g} = \begin{cases} 1 & \text{if} \quad l \in L_c \wedge l \in L_g \\ 0 & \text{if} \quad l \notin L_c \vee g \notin (G_{PE,c} \cup G_{PP,c}) \vee \\ & \quad\quad (l \in L_{SYS} \wedge l \notin L_{PE}) \vee \\ & \quad\quad (l \notin L_{lp} \wedge g \in G_{PP}) \\ X & \text{else} \end{cases} \quad (4)$$

$$f_D = \sum_{PE\in G_{PE}} \sum_{l\in L} \sum_{m\in M_l} \sum_{s\in S_m} \frac{ET_{m,PE}}{\phi_{s,PE}\cdot f_{PE}} \cdot b_{l,PE} \cdot \overbrace{\left(\sum_{\tilde{l}\in L} \sum_{\tilde{m}\in M_l} \sum_{\tilde{s}\in S_{\tilde{m}}} \phi_{\tilde{s},PE} \cdot b_{\tilde{l},PE} \right)}^{U_{PE}} + \sum_{(l_{src},l_{dst})\in L^2} \sum_{(g_{src},g_{dst})\in (G_{PE}\cup G_{PP})^2} b_{l_{src},g_{src}} \cdot b_{l_{dst},g_{dst}} \cdot |S_{l_{src},l_{dst}}| \cdot \tilde{D}(g_{src},g_{dst}) \tag{5}$$

$$f_E = \sum_{PE\in G_{PE}} \sum_{l\in L} \sum_{m\in M_l} |S_m| \cdot E_{m,PE} \cdot b_{l,PE} + \sum_{(l_{src},l_{dst})\in L^2} \sum_{(g_{src},g_{dst})\in (G_{PE}\cup G_{PP})^2} b_{l_{src},g_{src}} \cdot b_{l_{dst},g_{dst}} \cdot |S_{l_{src},l_{dst}}| \cdot \tilde{E}(g_{src},g_{dst}) \tag{6}$$

A layer must only be assigned to one PE:	$\sum_{PE\in G_{PE}} b_{l,PE} = 1 \;\forall\, l \in L$
PEs must not be overloaded:	$\sum_{l\in L} \sum_{m\in M_l} \sum_{s\in S_m} b_{l,PE} \cdot \phi_{s,PE} \le 1 \;\forall\, PE \in G_{PE}$
Uplinks to SBs must not be overloaded:	$\sum_{(l_{src},l_{dst})\in L^2} \sum_{s\in S_{l_{src},l_{dst}}} \phi_{s,h} \cdot b_{l_{src},PE(h)} \cdot (1 - b_{l_{dst},PE(h)}) \le 1 \;\forall\, h \in H_{up}$
Downlinks from SBs must not be overloaded:	$\sum_{(l_{src},l_{dst})\in L^2} \sum_{s\in S_{l_{src},l_{dst}}} \phi_{s,h} \cdot b_{l_{dst},PE(h)} \cdot (1 - b_{l_{src},PE(h)}) \le 1 \;\forall\, h \in H_{down}$
Flit buffer in SBs must not be exceeded:	$\sum_{PE_{src}\in G_{PE,SB}} \sum_{(l_{src},l_{dst})\in L^2} \sum_{s\in S_{l_{src},l_{dst}}} 2q \cdot b_{l_{src},PE_{src}} \cdot (1 - b_{l_{dst},PE_{src}}) \le Q_{SB,max} \;\forall\, SB \in G_{SB}$

$H_{\{up,down\}}$	Contains the uplinks / downlinks of system	$\{\cdot\}^2$	Returns a set of all possible pairs of the argument
$Q_{SB,max}$	Available buffer space in SB	$S_{l_{src},l_{dst}}$	Set of flow segments that connect l_{src} with l_{dst}
$\lvert\cdot\rvert$	Number of elements in the set	$\{ET,E\}_{m,PE}$	Execution cycles / energy to execute m on PE
S_m	Contains the flow segments that are processed by m (including virtual segments)	$G_{PE,SB}$	Contains the PEs whose uplinks are connected to SB
		$PE(h)$	Returns the processing engine that is connected to h

Table 1: The ILP to map the networking application on the multiprocessor

where X denotes that the variable has no fixed value and l_{SYS} contains the layers that have already been assigned in a previous optimization step (including logical ports). We use two cost functions to either minimize delay per packet (cf. eq. 5) or energy per packet (cf. eq. 6). Both functions describe the resource consumption for executing the packet methods on PEs and for forwarding flow segments between PEs. The first part of the delay cost function has been derived from eq. 1 and eq. 2. The functions $\{\tilde{D},\ \tilde{E}\}(PE_{src}, PE_{dst})$ return an approximation of the energy consumption resp. delay caused if PE_{src} communicates with PE_{dst}. To obtain these values, we use a shortest path algorithm where the links are weighted with the resp. resource (energy or delay for forwarding a maximum-sized packet).

The set of constraints for the ILP are also summarized in table 1. They define the technology limits. Please note that the buffer constraint reflects a shared memory SB. It has to be changed if separate buffers are utilized. We do not consider the buffer size inside a PE as a constraint since we assume that the PE has enough local memory to hold the program code, data and the packet queues. We obtain the mapping of layers to PEs by minimizing the cost function subject to the constraints with the 0-1 optimization problem solver "opbdp" [1]. It also provides an automatic linearization of the quadratic terms. The mapping time for one system instance is below one minute for 16 PEs and a flow processing graph consisting of 26 layers and 88 flow segments on a Pentium 4 (1.6 GHz).

In the last optimization step, the connections between the modules are routed through the NoC. For each flow segment s, a set of links H_s is obtained that are used to forward the flow segments. If the constraints of one of the ILPs can not be satisfied or there are not enough routing resources left, the mapping fails. Otherwise, it is guaranteed that the system achieves the throughput defined by the port data rate B_{lp}.

5. An exemplary application

We are using a component-based resource consumption model based on the techniques presented in [11] and our previous work about NoCs [2] and processing engines [5]. Besides the discussed resources, it is able to estimate the area and power requirements of the system. The target technology was a $130\,\mathrm{nm}$ process from UMC that allows a clock frequency of $f_{PE} = 230\,\mathrm{MHz}$. The estimates for the memory area and power consumption were collected from data sheets from Virage Logic ($2\,\mathrm{kB}$ flit buffer for the SBs) and MoSys ($128\,\mathrm{kB}$ 1T-SRAM-Q for the PEs). The area and power for the functional units of the PEs are based on [5]. To demonstrate the feasibility of our approach, we have analyzed the system outlined in fig. 2. For the experiments, we have iterated through a set of configurations, denoted by a configuration string: <optimization(**E**nergy/**D**elay)>*<port

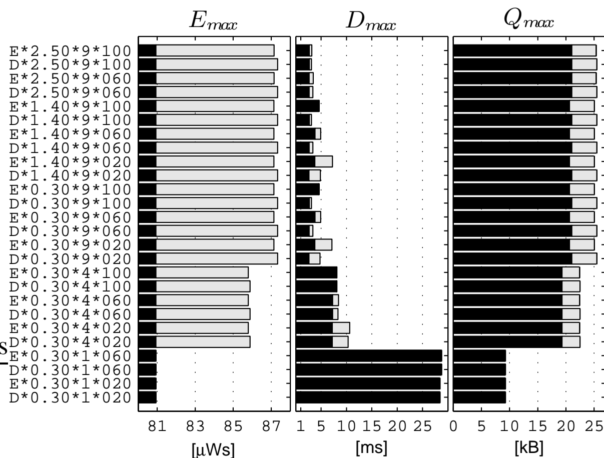

Figure 4: Results of the design space exploration

data rate B_{lp} [Mbps]>*<number of PEs $|G_{PE}|$>*<link data rate B_h [Mbps]>.

Figure 4 shows the results. The energy E_{max} is the maximum energy consumed per packet for processing and forwarding all flow segments. The delay D_{max} is the maximum possible duration that a packet requires to pass through a packet flow pf. Finally, Q_{max} is the maximum buffer space required for the internal communication. The black areas denote the contribution of processing and the grey areas mark the contribution of communication to the resource consumption. The diagram shows that both the energy and delay optimization work as expected. One PE ($1.6\,\mathrm{mm}^2$ area) and a 2x2 grid of PEs ($10.8\,\mathrm{mm}^2$ area) can only process the flow graph at the $0.3\,\mathrm{Mbps}$ port data rate. All other data rates overload the PEs. A 3x3 PE grid has a maximum throughput of $2.5\,\mathrm{MBit}$ per port. It has an approximated area consumption of $25.5\,\mathrm{mm}^2$ and a power consumption below $181\,\mathrm{mW}$. The largest system has also the smallest delay because more less loaded PEs are available. At a link data rate above $60\,\mathrm{MBit}$, the forwarding of flow segments between PEs does not have an effect on the worst case delay any longer since the processing becomes the dominating factor. The most energy and delay is consumed inside the PEs that have also the biggest impact on the delay. The total buffer usage increases slightly if delay is minimized due to the additional communication.

6. Conclusion

We have introduced a novel design method for finding a mapping of a network protocol to different configurations of a multiprocessor SoC in an automated way. Our approach is based on a hierarchical optimization model solved by applying integer linear programming. It is able to minimize the energy or delay per packet while meeting the technological constraints of the system. We have also shown how a non-blocking scheduling strategy, based on generalized processor sharing (GPS), can reduce the buffer requirements to tackle an important design problem of NoCs. A first exemplary application has shown that the NoC has a neglectable impact on the delay and energy per packet for the considered network protocol. It became also evident that a 3x3 multiprocessor can be applied in mobile applications due to its low power requirements. In our future work, we want to analyze the effect of NoCs with different connection topologies and router characteristics. We will also implement a multiprocessor to prove our concepts and to determine the accuracy of our estimation methods.

References

[1] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, January 1995.

[2] A. Brinkmann, J.-C. Niemann, I. Hehemann, D. Langen, M. Porrmann, and U. Rückert. On-Chip Interconnects for Next Generation System-on-Chips. In *Proc. of the 15th Annual IEEE Int. ASIC/SOC Conference*, September 2002.

[3] P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk. *Network Processor Design: Issues and Practices*, volume 1. Morgan Kaufmann Publishers, 2002.

[4] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proc. of the Design Automation Conference*, Las Vegas, USA, June 18-22 2001.

[5] M. Grünewald, J.-C. Niemann, and U. Rückert. A performance evaluation method for optimizing embedded applications. In *Proc. of the 3rd IEEE Int. Workshop on System-On-Chip for Real-Time Applications*, pages 10–15, Calgary, Alberta, Canada, June 30 - July 2 2003.

[6] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *In Proc. of Design, Automation and Test in Europe*, pages 250 – 256, 2000.

[7] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to fbw control in integrated service networks: The single node case. *IEEE / ACM Transactions on Networking*, 1-3:344–357, June 1993.

[8] S. Prakash and A. C. Parker. Synthesis of Application-Specifi c Multiprocessor Systems including Memory Components. *Journal of VLSI Signal Processing*, 1994.

[9] E. Rijpkema, K. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *Proc. of Design, Automation and Test in Europe*, 2003.

[10] S. Rührup, C. Schindelhauer, K. Volbert, and M. Grünewald. Performance of distributed algorithms for topology control in wireless networks. In *Proc. of the Int. Parallel and Distributed Processing Symposium*, Nice, France, 22 - 26 2003.

[11] T. Šimunić, L. Benini, and G. De Micheli. Energy-effi cient Design of Battery-Powered Embedded Systems. *Special Issue of IEEE Transactions on VLSI*, 2001.