Design Space Exploration of Network Processor Architectures

Lothar Thiele, Samarjit Chakraborty, Matthias Gries, Simon Künzli
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zürich, Switzerland
{thiele|samarjit|gries|kuenzli}@tik.ee.ethz.ch

Abstract

We describe an approach to explore the design space for architectures of packet processing devices on the system level. Our method is specific to the application domain of packet processors and is based on (1) models for packet processing tasks, a specification of the workload generated by traffic streams, and a description of the feasible space of architectures including computation and communication resources, (2) a measure to characterize the performance of network processors under different usage scenarios, (3) a new method to estimate end-to-end packet delays and queuing memory, taking task scheduling policies and bus arbitration schemes into account, and (4) a evolutionary algorithm for multi-objective design space exploration. Our method is analytical and based on a high level of abstraction, where the goal is to quickly identify interesting architectures, which may be subjected to a more detailed evaluation, e.g. using simulation. The feasibility of our approach is shown by a detailed case study, where the final output is three candidate architectures, representing different cost versus performance tradeoffs.

1. Introduction

Network Processors usually consist of multiple processing units such as CPU cores, micro-engines, and dedicated hardware for compute-intensive tasks such as header parsing, table look-up and encryption/decryption. Together with these, there are also memory units, caches, interconnections, and I/O interfaces. Following a system-on-a-chip (SoC) design method, these resources are put on a single chip and must interoperate to perform packet processing tasks at line speed. The process of determining the optimal hardware and software architecture for such processors is faced with issues involving resource allocation and partitioning, and the architecture design should take into account different packet processing functions, task scheduling options, information about the packet forms, and the QoS guarantees that the processor should be able to meet. The available chip area for putting the different components together might be restricted, imposing additional constraints. Further, network processors may be used for many different application scenarios such as those arising in backbone and access networks. Whereas backbone networks can be characterized by very high throughput demands but relatively simple processing requirements per packet, access networks show lower throughput demands but high computational requirements for each packet. The architecture exploration and evaluation of network processors therefore pose many interesting challenges and involve many tradeoffs and a complex interplay between hardware and software.

There are several characteristics which are specific to the packet processing domain, and these do not arise in other application areas such as classical digital signal processing (although both domains involve the processing of event streams). The packet processing case is concerned with the processing of interleaved streams of data packets, where for each packet stream a certain sequence of tasks must be executed (so there are usually no recurrent or iterative computations), the tasks are of high granularity, and they are often scheduled dynamically at run-time. Due to this difference with other known target domains for system-level design space exploration, several new questions arise: How should packet streams, task structures and hardware and software resources appropriately be modelled? How can the performance of a network processor architecture be determined in the case of several (possibly conflicting) usage scenarios? Since the design space can be very large, what kind of strategy should be used to efficiently explore all options and to obtain a reasonable compromise between various conflicting criteria?

In this paper we present a framework for the design space exploration of embedded systems operating on packet streams where we address the above issues. The underlying principles of our approach can be outlined as follows:

- Our framework consists of a task and a resource model, and a *real-time calculus* [2, 17] for reasoning about packet streams and their processing. The task model represents the different packet processing functions such as header processing, encryption, processing for special packets such as voice and video, etc. The resource model captures the information about different available hardware resources, the possible mappings of packet processing functions to these resources, and the associated costs. There is also the information about different traffic flows, which are specified using their *arrival curves* [5] and possible deadlines.

- The design space exploration is posed as a multi-objective optimization problem. There are different conflicting criteria such as chip area, on-chip memory requirements, and performance (such as the throughput and the number of flow classes that can be supported). The output is a set of different hardware/software architectures representing the different tradeoffs.

- Given any architecture, the calculus associated with the framework is used to analytically determine properties such as delay and throughput experienced by different flows, taking into consideration the underlying scheduling disciplines at the different resources. An exploration strategy comes up with possible alternatives from the design space, which are evaluated using our calculus, and the feedback guides further exploration.

To speedup the exploration, unlike previous approaches we use several linear approximations in the real-time calculus, so that the different system properties can be quickly estimated. We also show how different resources with possibly different scheduling strategies, and communication resources with different arbitration mechanisms can be combined together to construct a *scheduling network*, which allows to determine, among other things, the size of shared as well as per-resource memory. Our multi-objective design space exploration takes into account the fact that there can be different scenarios in which the processor may be deployed, and this is modelled in the form of different *usage scenarios*. Lastly, the way we allocate the multiple processing units and the memory units, our optimization strategy also optimizes the load balancing between them.

Related work All the previous work on design space exploration of network processors (such as [4] and [18]) relied on simulation techniques, where different architectures are simulated and evaluated using benchmark workloads. When the search space being explored is large, it might be too expensive to evaluate all the alternatives using simulation. In contrast to this, different architectures in our framework can be analytically evaluated to determine bounds on resource requirements (such as memory and cache sizes), and QoS parameters (such as delay experienced by packets). The focus here is on a high level of abstraction, where the goal is to quickly identify interesting architectures which can be further evaluated (for example by simulation) taking lower level details into account.

Recent research on packet processors has dealt with task models [16], task scheduling [13], operation system issues [12], and packet processor architectures [8, 15]. All of these issues collectively play a role in different phases of the design space exploration of such devices, and the relevant ones in the context of our abstraction level have been considered in this paper.

An attempt to perform system-level design space exploration of network and packet processors has been described in [16]. Here, the exploration is performed by an integer linear program and the estimation of the system properties is limited to very simple linear models. The complexity of the underlying optimization problem prevents the use of this method for realistic design problems. Moreover, the memory requirements are only analyzed for a shared memory architecture and the overhead for communication between computational resources is not considered at all.

Related work on the design space exploration of SoC communication architectures (especially in the context of the case study that we present in this paper) include [10, 9] (and the references therein). However, in contrast to our approach, the methods used in these papers largely rely on simulation.

The next section formally describes the task and the resource structures, following which we describe the framework for analytically evaluating prospective candidate architectures in Section 3. Section 4 describes techniques for the multiobjective design space exploration, and a case study illustrating our methodology is presented in Section 5.

2. Models for Streams, Tasks, and Resources

In the following, models for the workload generated by streams, for the typical task structure associated with the packet processing of streams, and for the definition of architectures are described.

2.1. Workload generated by Packet Streams

A network processor operates on interleaved streams of packets which enter the device. In order to determine the load of the processing device, it is necessary to know the number of packets arriving per time unit. This information can be formalized using arrival curves which allow us to derive deterministic bounds on the workload and which are common in the networking community, see, for instance, the T-SPEC model [14] of the IETF.

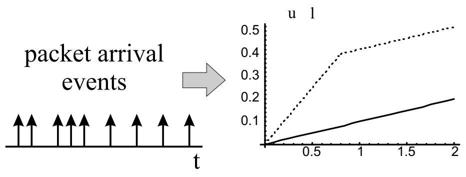

Figure 1. Representation of arrival curves.

Definition 1 (Arrival Curves) *For any $\Delta \in \mathbb{R}_{\geq 0}$, the lower arrival curve $\alpha^l_f(\Delta)$ of a stream f is smaller than the number of packets arriving in any time interval of length Δ in the stream f. In a similar way, the maximum number of packets in any interval of length Δ is always smaller than the upper arrival curve $\alpha^u_f(\Delta)$, i.e. $\alpha^u_f(\Delta), \alpha^l_f(\Delta) \in \mathbb{R}_{\geq 0}$.*

Arrival curves may be determined by service level agreements (e.g. a T-SPEC), by analysis of the traffic source (e.g. a sensor), or by traffic measurement. Fig. 1 shows an example of an arrival curve.

All packets belonging to the same stream are processed in the same way, i.e. a constant set of tasks is executed in a defined order. This task structure is characteristic for packet processing and can be described as follows.

Definition 2 (Task Structure) *We define a set of streams F and a set of tasks T. To each stream $f \in F$ there is associated a directed acyclic graph $G(f) = (V(f), E(f))$ with task nodes $V(f) \subseteq T$ and edges $E(f)$. The tasks $t \in V(f)$ must be executed for each packet of stream f while respecting the precedence relations in $E(f)$.*

Tasks associated with different streams can be combined into one conditional task graph where depending on the stream to which a packet belongs, the packet takes different paths through this graph. See Fig. 10 for an example.

Definition 3 (Requests) *To each stream $f \in F$ there is associated an end-to-end deadline $d_f : F \rightarrow \mathbb{R}_{\geq 0}$. If a task t can be executed on a resource s, then it creates a "request", i.e. for all possible task to resource bindings there exist a request $w(t, s) \in \mathbb{R}_{\geq 0}$.*

The end-to-end deadline d_f denotes the maximally allowed time span from the arrival of any packet of stream f to the end of the execution of the last task for that packet. A reasonable unit for a request of a task on a computing resource may be the number of cycles or instructions.

2.2. System Architecture

Network processors are heterogeneous in nature. On dedicated or application specific instruction set components, simple tasks with high data rate requirements are

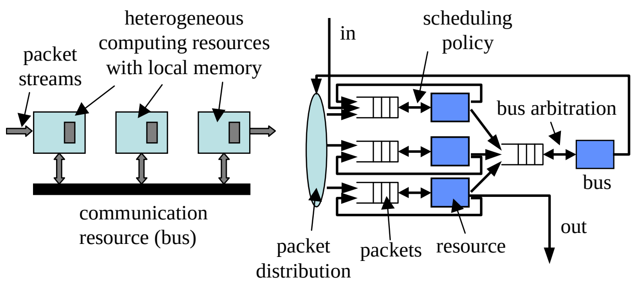

Figure 2. Example of a physical (left) and logical (right) structure of a network processor architecture.

executed. Longer and more complicated execution chains are transferred to software tasks running on (homogeneous) multiprocessors. In this case, usually run-time scheduling methods are used in order to fairly share the available resources to the packets belonging to the different streams, see [3]. Each computation resource may make use of dedicated local memory such as on-chip embedded memory. Depending tasks executed on the same resource do not suffer from a communication overhead, whereas tasks on different resources must communicate through a communication resource (a bus). We want to point out that the task structure defined in Def. 2 may therefore also contain tasks which represent communication. A communication request may be specified by the number of bytes of the transfer. The introduction of communication tasks requires the knowledge of the bindings of tasks to resources. How this is incorporated into our framework in a transparent way for the user will be described in Section 3.2. The consideration of buses and local memories allows a more realistic representation of typical network processor architectures than the models described in related work [16]. A sketch of a heterogeneous architecture with different packet paths is shown in Fig. 2 on the left.

Therefore, our model of a feasible system architecture consists of (1) available resource types including their processing or communication capability and performance described by service curves, (2) costs for implementing a resource on the network processor, and (3) the scheduling/arbitration policies and associated parameters. The logical structure of a system architecture is shown on the right of Fig. 2. Here we see that the processing components have associated memories which store packets waiting for the next task to be executed on them. A corresponding scheduling policy selects a packet and starts the execution. The processing of the current packet may be preempted in favor of a task for another packet. After the execution of a task a packet may be reinserted into the input queue of the current resource or may be redistributed to another resource using

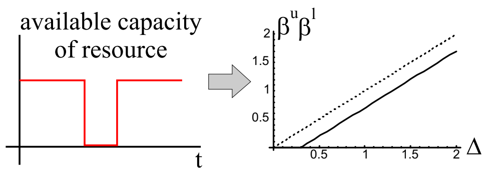

Figure 3. Representation of service curves.

a bus. Without restricting the applicability of our approach we limit the description of suitable architectures to a single bus to simplify the explanation.

Service curves are defined in a similar way than arrival curves. They describe bounds on the computation or communication capability of the available resource components. If a resource is loaded with the execution of some tasks, it is clear that the available computing power in an interval may vary and thus the time for executing a further task. Fig. 3 shows an example.

Definition 4 (Service Curves) *For any $\Delta \in \mathbb{R}_{\geq 0}$ and any resource type $s \in S$, the lower service curve $\beta^l_s(\Delta)$ is smaller than the number of available computing/communication units in any time interval of length Δ. In a similar way, the maximum number of computing/communication units in any interval of length Δ is always smaller than the upper service curve $\beta^u_s(\Delta)$.*

Definition 5 (Resources) *We define a set of resource types S. To each type $s \in S$ there is associated a relative implementation cost $cost(s) \in \mathbb{R}_{\geq 0}$ and the number of available instances $inst(s) \in \mathbb{Z}_{\geq 0}$. To each resource instance there is associated a finite set of scheduling policies $sched(s)$ which the component supports, a lower service curve β^l_s and an upper service curve β^u_s.*

Definition 6 (Task to Resource Mapping) *The mapping relation $M \subseteq T \times S$ defines possible mappings of tasks to resource types, i.e. if $m = (t, s) \in M$ then task t could be executed on resource type s.*

If $(t, s) \in M$ allows the execution of task t on resource s, a request $w(t, s) \in \mathbb{R}_{\geq 0}$ is associated with this mapping (see Def. 3).

3. Analysis using Scheduling Network

Although we have chosen a particularly simple cost model, it is not obvious how to determine the maximum number of stored packets or the maximum end-to-end delays since all packet streams share common resources. For example, the computation time for a task t depends on its request $w(t, s)$, on the available processing power of the resource, i.e. β^l_s and β^u_s, and on the scheduling policy applied. In addition, as the packets may flow from one resource to the next one, there may be intermediate bursts and packet jams. It is interesting to see, that there nevertheless exists a computationally efficient possibility to derive provably correct bounds on the end-to-end delays of packets and the required memory.

We exploit the fact that characteristic chains of tasks are executed for packet streams and that streams are processed independently. Based on this knowledge we are able to construct a scheduling network where the real-time calculus is applied from node to node in order to derive deterministic bounds. Note that the execution of constant chains of tasks is one of the major characteristics in the network processing domain which cannot be found in any other domain.

Basis for the determination of end-to-end delays and memory requirements is the description of packet streams in communication networks by using a network calculus, see [5]. Recently, this approach has been re-formulated in an algebraic setting, see [2]. In [17], a comparable approach has been used to describe the behavior of processing resources.

3.1. Building Blocks of the Scheduling Network

The basic idea of the following performance estimation is the provision of a network theory view of the system architecture. In particular, packet streams and resource streams flow through a network and thereby adapt their arrival and service curves, respectively. Inputs to a network node are arrival curves of packet streams and service curves of a resource. The outputs describe the resulting arrival curves of the processed packet streams and the remaining service curves of the (partly) used resource. These resulting arrival and service curves can then serve as inputs to other nodes of the scheduling network.

In order to understand the basic concept, let us describe a very simple example of such a node, namely the preemptive processing of packets of one stream by a single processing resource. Following the discussion of Fig. 2, a packet memory is attached to a processing resource which stores those packets that have to wait for being processed. In [16], the following Theorem has been derived which describes the processing of a packet stream in terms of the already defined arrival and service curves.

Theorem 1 *Given a packet stream described by the arrival curves α^l and α^u and a resource stream described by the service curves β^l and β^u. Then the following expressions bound the remaining service of the resource node and the arrival curve of the processed packet stream:*

$$\alpha^{l'}(\Delta) = \inf_{0 \leq u \leq \Delta} \left\{ \alpha^l(u) + \beta^l(\Delta - u) \right\} \quad (1)$$

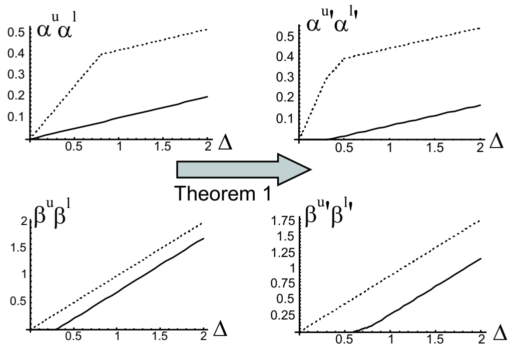

Figure 4. Remaining service and resulting arrival curve according to Theorem 1.

$$\alpha^{u\prime}(\Delta) = \inf_{0\le u\le\Delta}\{\sup_{v\ge 0}\{\alpha^u(u+v)-\beta^l(v)\} + \beta^u(\Delta-u),\beta^u(\Delta)\} \quad (2)$$

$$\beta^{l\prime}(\Delta) = \sup_{0\le u\le\Delta}\{\beta^l(u)-\alpha^u(u)\} \quad (3)$$

$$\beta^{u\prime}(\Delta) = \sup_{0\le u\le\Delta}\{\beta^u(u)-\alpha^l(u)\} \quad (4)$$

Note that the arrival curve as used above describes bounds on the *computing request* and *not* on the *number of packets*. In Fig. 4, an example for remaining arrival and service curves is given. As we deal with packet streams in the system architecture, we need to convert packets to computing requests. Given bounds on a packet stream of the form $[\overline{\alpha}^l,\overline{\alpha}^u]$ we can determine bounds on the related computing requests

$$[\alpha^l,\alpha^u] = [w\overline{\alpha}^l, w\overline{\alpha}^u] \quad (5)$$

considering the request w for each packet. The notation $[\alpha^l,\alpha^u]$ represents the fact that α^l and α^u are lower and upper curves of the same stream.

The conversion of the output stream is more involved, as we usually suppose that a next component can start processing after the whole packet arrived:

$$[\overline{\alpha}^{l\prime},\overline{\alpha}^{u\prime}] = [\lfloor\alpha^{l\prime}/w\rfloor,\lceil\alpha^{u\prime}/w\rceil] \quad (6)$$

The whole transformation is depicted in Fig. 5.

The extension to the execution of several tasks of different streams by a resource according to a fixed priority scheduling policy is described in the following example. Other scheduling policies can be described in a similar way.

Example 1 (Fixed Priority Scheduling) *We can simply describe one of the major scheduling approaches used in current network processors and general real-time systems, namely fixed priority preemptive scheduling, see [3]. In this case, to each input stream $f \in F$ there is assigned a fixed priority $prio(f) \in \mathbb{Z}_{\ge 0}$. At any point in time, a processing device operates on the packet in its memory (see Fig. 2,*

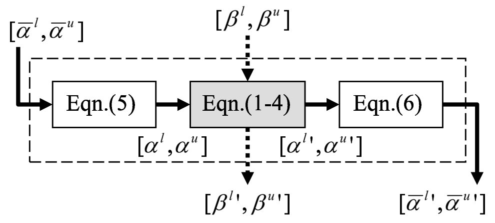

Figure 5. Block diagram showing the transformation of packet and resource streams by a processing device. The dotted arrows represent the resource flow, the others show the flow of packets and requests.

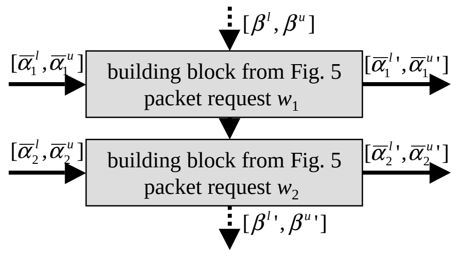

Figure 6. Representation of fixed priority preemptive scheduling of packet streams on a single processing resource. Stream 2 has a smaller priority than 1.

scheduling policy) which has the highest priority. If there are packets with the same priority, they are served following an FCFS (first come first serve) strategy. This can be modelled directly using the basic building block shown in Fig. 5. In Fig. 6, a simple example with just two input streams described by their arrival curves $[\alpha_1^l,\alpha_1^u]$ and $[\alpha_2^l,\alpha_2^u]$ and packet computing requests w_1 and w_2 is depicted.

3.2. Scheduling Network Construction

The scheduling network which enables the estimation of performance such as end-to-end deadlines and memory consumption is constructed using the specification data in Section 2 and a specification of the system architecture. In order to simplify the explanation, we restrict ourselves to the use of a fixed priority scheduling policy for all resource types, i.e. $sched(s) = \{\text{fixedPriority}\}$ for all resource types $s \in S$. The basic idea is that the specifications of the packet streams pass from one resource to the next one. The order is determined by the precedence relations in $E(f)$ in the task structure, see Def. 2, and the binding of tasks to resources. The resource flows, i.e. the capabilities of the resources, also pass through the network. The order is mainly determined by the priorities assigned to packet streams.

Definition 7 (System Architecture) *The allocation of resources can be described by the function $alloc(s) \in \mathbb{Z}_{\geq 0}$ which denotes the number of allocated instances of resource type s. The binding of tasks $t \in T$ to resources is specified by a relation $B \subseteq T \times S \times \mathbb{Z}_{\geq 0}$, i.e. if $b = (t, s, i) \in B$ with $1 \leq i \leq alloc(s)$ then task t is executed on the ith instance of resource type s. The scheduling policy is described by a function $prio(f) \in \mathbb{Z}_{\geq 0}$ which associates a priority to each stream f in a usage scenario.*

Note that a system architecture is not only described by the type and number of resource components but also by the binding of tasks to those components. This mapping may depend on the stream in which the task is active and on the scenario under which the system architecture is evaluated.

The model combines two different forms of parallelism. On the one hand it is possible to have dedicated hardware modules for certain tasks. The resulting architecture is heterogeneous. On the other hand, we may have parallel resource instances of equal type ($alloc(s) > 1$) which may process complete packet streams.

Now, we can describe the construction of a scheduling network for a given scenario. Note, that in general we have different networks for each usage scenario as the tasks, streams, and priorities may be different. Assuming that the user only specifies computation tasks since the definition of communication tasks mapped to communication resources requires the knowledge of a valid binding of the computation tasks on resources, a preparation step is required to introduce communication into our scheduling network. Again, we limit our description to a single bus.

- (preparation to include communication) For all flows f and all task dependencies $(t_i, t_j) \in E(f)$, if t_i and t_j are not bound to the same resource instance, add a communication task t_c to $V(f)$ and the edges (t_i, t_c), (t_c, t_j) to $E(f)$. Remove edge (t_i, t_j) from $E(f)$ and bind t_c to the communication resource $s_c \in S$.

- Include in the scheduling network one source resource node and one target resource node for each allocated instance of resource type $s \in S$. Include in the scheduling network one source packet node and one target packet node for each stream f present in the scenario.

- Construct an ordered set of tuples T_f which contains (t, f) for all streams f in the scenario and for all tasks $t \in V(f)$ in this stream. Order these tuples according to the priorities of the corresponding streams and according to the precedence relations $E(f)$. For each tuple (t, f) in T_f, add a scheduling node corresponding to that shown in Fig. 5 to the scheduling network.

- For all streams f in the scenario we add the following connections to the scheduling network:

  For all task dependencies $(t_i, t_j) \in E(f)$ the packet stream output of scheduling node (t_i, f) is connected to the packet stream input of (t_j, f).

  For each resource instance of any type $s \in S$, consider the scheduling nodes (t, f) where the task t is bound to that instance of s. If $(t_i, f) \prec (t_j, f)$ in the ordered set T_f, then connect the resource flow output of (t_i, f) to the resource flow input of (t_j, f).

As a result of applying this algorithm we get a scheduling network for a scenario containing source and target nodes for the different packet streams and resource flows as well as scheduling nodes which represent the computations described in Fig. 5. An example is given in Fig. 14.

Given the arrival curves for all stream source nodes, i.e. $[\alpha_f^l, \alpha_f^u]$ for all streams f in a scenario, and the initial service curves for the allocated resource instances, i.e. $[\beta_{s,i}^l, \beta_{s,i}^u]$ for resource type s with $1 \leq i \leq alloc(s)$ allocated resources, we can determine the properties of all internal packet streams and resource flows. It remains to be seen, how we can determine the end-to-end delays of packets and the necessary memory.

3.3. System Properties

In order to estimate the properties of the system architecture for a network processor we need quantities like bounds on end-to-end delays of packets and memory requirements. Using well known results from the area of communication networks, see e.g. [5], the bounds derived in Theorem 1 can be used to determine the maximal delay of events and the necessary memory to store waiting events.

$$delay \leq \sup_{u \geq 0} \left\{ \inf\{\tau \geq 0 : \alpha^u(u) \leq \beta^l(u + \tau)\} \right\} \quad (7)$$

$$backlog \leq \sup_{u \geq 0} \{\alpha^u(u) - \beta^l(u)\} \quad (8)$$

In other words, the delay can be bounded by the maximal horizontal distance between the curves α^u and β^l whereas the backlog is bounded by the maximal vertical distance between them.

In case of the scheduling network constructed above, we need to know which curves to use in (7) and (8). The upper arrival curve is that of an incoming packet stream, i.e. α_f^u of the investigated stream f in the current scenario. The service curve β^l to be used in (7) and (8) is the *accumulated* curve of all scheduling nodes through which the packets of stream f pass in the current scenario. As has been described in e.g. [2], this quantity can be determined through an iterated convolution. To this end, let us suppose that the packets of stream f pass through scheduling nodes $p_1, ..., p_m$ which have the lower service curves $\beta_1^l, ..., \beta_m^l$ at their resource flow inputs. Then β^l in (7) and (8) can be determined using the following recursion:

$$\overline{\beta}^l_1 = \beta^l_1 \quad (9)$$
$$\overline{\beta}^l_{i+1} = \inf_{0\le u\le\Delta}\left\{\overline{\beta}^l_i(u) + \beta^l_{i+1}(\Delta - u)\right\} \forall i > 1 \quad (10)$$
$$\beta^l = \overline{\beta}^l_{m+1} \quad (11)$$

As a result, we can compute bounds on the maximum delay *delay* and maximum shared memory *backlog* for a stream f by use of the scheduling network as constructed above. If we are interested in the memory requirements for an implementation with separated local memories as shown in Fig. 2, we can generate the accumulated service curves for all sequences of tasks which stay on the same resource instance. There are several special cases, where we can make use of an accumulated service curve to determine tighter bounds than by independently deriving memory requirements for each node. For instance, suppose that a packet stream is processed first on a general-purpose component. For a certain task the stream is then delegated to a more specialized unit. After being processed on that dedicated resource instance, the stream returns to the former component. An analysis using the accumulated service curve over all processing steps including the ones on the specialized unit may derive tighter memory bounds for the general-purpose component than two independent analyses of the first and the second visit of the stream at that resource. We will not describe all subcases here because the form of the equations (7) to (11) is not affected.

The memory requirements derived by an analysis of communication resources must be assigned to the corresponding sending task (and therefore to the resource instance bound to that task). This memory requirement is visualized as an output queue before the communication resource in Fig. 2 on the right.

3.4. Piecewise Linear Approximation

Clearly, the equations used in Theorem 1 are expensive to compute. It may also be noted that this set of equations has to be computed for all the scheduling nodes in a scheduling network. Moreover, when the design space exploration is based on evolutionary multiobjective algorithms, the performance of many system architectures need to be estimated, and there might be several usage scenarios per system architecture.

To overcome this computational bottleneck, we propose a piecewise linear approximation of all arrival and service curves. Based on this, all the equations in Theorem 1 can be efficiently computed using symbolic techniques. Due to space restrictions, we only describe the basic concepts here and give a few simple examples. Fig. 7 shows how the arrival and service curves look like when each curve is approximated by a combination of two line segments.

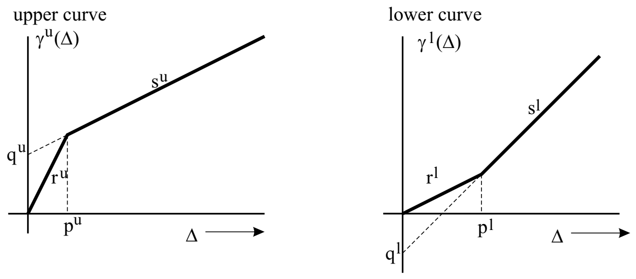

Figure 7. Simple representation of upper and lower curves.

In this case, we can write:

$$\gamma^u(\Delta) = \min\{r^u\Delta, q^u + s^u\Delta\},\ \gamma^l(\Delta) = \max\{r^l\Delta, q^l + s^l\Delta\}$$

where,

$$q^u \ge 0,\quad r^u \ge s^u \ge 0,\quad r^u = s^u \Leftrightarrow q^u = 0$$
$$q^l \le 0,\quad 0 \le r^l \le s^l,\quad r^l = s^l \Leftrightarrow q^l = 0$$

As a shorthand notation we denote curves γ^u and γ^l by the tuples $U(q,r,s)$ and $L(q,r,s)$, respectively. An example of a piecewise linear approximation of the remaining lower service curve $\beta^{l'}(\Delta)$ in Theorem 1 is given next.

Theorem 2 *Given arrival curves and service curves $\alpha^u = U(q_\alpha, r_\alpha, s_\alpha)$, $\beta^l = L(q_\beta, r_\beta, s_\beta)$. Then the remaining lower service curve can be approximated by the curve*

$$\beta^{l'} = L(q,r,s)$$

where

$$q = \begin{cases} q_\beta - q_\alpha & \text{if } s_\alpha \le s_\beta \\ 0 & \text{if } s_\alpha > s_\beta \end{cases}$$
$$r = \max\{r_\beta - r_\alpha, 0\}$$
$$s = \max\{s_\beta - s_\alpha, 0\}$$

Proof. To see that $L(q,r,s)$ is a valid lower curve for the remaining service curve, it may be shown that

$$L(q,r,s)(\Delta) \le \sup_{0\le u\le\Delta}\{\beta^l(u) - \alpha^u(u)\}.$$

Note that $\beta^l(\Delta) - \alpha^u(\Delta)$ and also $\sup_{0\le u\le\Delta}\{\beta^l(u) - \alpha^u(u)\}$ are convex, since β^l and α^u are convex and concave respectively. Therefore, a valid lower bound can be determined by considering the two cases, $\Delta \to 0$ and $\Delta \to \infty$. If $\Delta \to 0$, we have

$$\sup_{0\le u\le\Delta}\{\beta^l(u) - \alpha^u(u)\} = \sup_{0\le u\le\Delta}\{r_\beta u - r_\alpha u\}$$

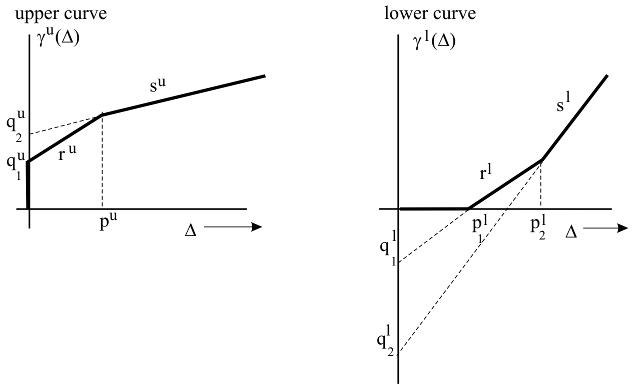

Figure 8. Improved approximation of upper and lower curves.

and therefore $r = \begin{cases} r_\beta - r_\alpha & \text{if } r_\beta > r_\alpha \\ 0 & \text{otherwise} \end{cases}$

If $\Delta \to \infty$ and $s_\beta > s_\alpha$ then

$$\sup_{0\leq u\leq\Delta}\{\beta^l(u) - \alpha^u(u)\} = \sup_{0\leq u\leq\Delta}\{q_\beta + s_\beta u - q_\alpha - s_\alpha u\}.$$

and therefore $s = \begin{cases} s_\beta - s_\alpha & \text{if } s_\beta > s_\alpha \\ 0 & \text{otherwise} \end{cases}$

$$q = \begin{cases} q_\beta - q_\alpha & \text{if } s_\beta > s_\alpha \\ 0 & \text{otherwise.} \end{cases}$$

□

All remaining equations on the curves can similarly be symbolically evaluated, including those which determine bounds on the delay and the backlog. Using these approximations, even for realistic task and processor specifications, hundreds of architectures can be evaluated within a few seconds of CPU time.

Improved approximations It should be noted here that it is possible to obtain improved approximations, for example by approximating the arrival and service curves by three linear segments instead of two, which was shown above. The resulting calculations however become more involved in this case. Fig. 8 shows the resulting arrival and service curves and, for instance, allows us to exactly model an arrival curve in the form of a T-SPEC [14]. In the case of an arrival curve, here q_1^u may represent the maximum workload because of a single packet, r^u can be interpreted as the burst rate and s^u the long term arrival rate.

The upper and the lower curves in this case can be written as:

$$\begin{aligned} \gamma^u(\Delta) &= \begin{cases} \min\{q_1^u + r^u\Delta, q_2^u + s^u\Delta\} & \text{if } \Delta > 0 \\ {q_1}^u & \text{if } \Delta = 0 \end{cases} \\ \gamma^l(\Delta) &= \max\{q_2^l + s^l\Delta, q_1^l + r^l\Delta, 0\} \end{aligned}$$

where,

$$\begin{array}{lll} q_2^u \geq q_1^u \geq 0, & r^u \geq s^u \geq 0, & r^u = s^u \Leftrightarrow q_1^u = q_2^u \\ q_2^l \leq q_1^l \leq 0, & 0 \leq r^l \leq s^l, & r^l = s^l \Leftrightarrow q_1^l = q_2^l \end{array}$$

The values of p^u and p_1^l, p_2^l (see Fig. 8) can be calculated as:

$$p^u = \begin{cases} \frac{q_2^u - q_1^u}{r^u - s^u} & \text{if } r^u > s^u \\ 0 & \text{if } r^u = s^u \end{cases}$$

$$p_1^l = \begin{cases} -\frac{q_1^u}{r^l} & \text{if } r^l > 0 \\ 0 & \text{if } r^l = 0 \end{cases}, \qquad p_2^l = \begin{cases} \frac{q_2^l - q_1^l}{r^l - s^l} & \text{if } r^l < s^l \\ p_1^l & \text{if } r^l = s^l \end{cases}$$

We denote the curves γ^u and γ^l in this case by $U(q_1, q_2, r, s)$ and $L(q_1, q_2, r, s)$ respectively.

Now given the upper arrival and the lower service curves respectively as:

$$\begin{aligned} \alpha^u &= U(q_{1\alpha}, q_{2\alpha}, r_\alpha, s_\alpha) \\ \beta^l &= L(q_{1\beta}, q_{2\beta}, r_\beta, s_\beta). \end{aligned}$$

The approximate remaining lower service curve as given by Theorem 2 can now be given in the form $\beta^{l'} = L(q_1, q_2, r, s)$. To compute q_1, q_2, r and s, we first compute p_α and $p_{1\beta}$ and $p_{2\beta}$. These are exactly similar to p^u, p_1^l and p_2^l which were computed above.

The ordering between p_α, $p_{1\beta}$ and $p_{2\beta}$ divides the real line $\mathbb{R}_{\geq 0}$ into four segments. Therefore, to compute the value of p_1 of the remaining service curve $\beta^{l'}$, it is sufficient to compute the value of $\beta^l(u) - \alpha^u(u)$ at $u = p_\alpha, p_{1\beta}$ and $p_{2\beta}$. For example, in the case where $p_{1\beta} \leq p_{2\beta} \leq p_\alpha$, when $u \leq p_{1\beta}$, $\beta^l(u) - \alpha^u(u)$ is certainly less than or equal to 0. When $u \in (p_{1\beta}, p_{2\beta}]$, then $\beta^l(u) - \alpha^u(u) = q_{1\beta} + r_\beta u - q_{1\alpha} - r_\alpha u$. When $u \in (p_{2\beta}, p_\alpha]$, then $\beta^l(u) - \alpha^u(u) = q_{2\beta} + s_\beta u - q_{1\alpha} - r_\alpha u$. Finally, when $u > p_\alpha$, then $\beta^l(u) - \alpha^u(u) = q_{2\beta} + s_\beta u - q_{2\alpha} - s_\alpha u$. From these four cases, we can compute the value of u for which $\beta^l(u) - \alpha^u(u) = 0$, and set p_1 to this value.

Now note that the curve $\beta^l(\Delta) - \alpha^u(\Delta)$ and also $\sup_{0\leq u\leq\Delta}\{\beta^l(u) - \alpha^u(u)\}$ is convex. Therefore, when $\Delta \to \infty$, $\sup_{0\leq u\leq\Delta}\{\beta^l(u) - \alpha^u(u)\} = \sup_{0\leq u\leq\Delta}\{q_{2\beta} + s_\beta u - q_{2\alpha} - s_\alpha u\}$. Hence, for the curve $\beta^{l'}$, $s = s_\beta - s_\alpha$ and $q_2 = q_{2\beta} - q_{2\alpha}$ if $s_\beta > s_\alpha$ and $s = 0$ and $q_2 = 0$ otherwise.

Therefore,

$$q_2 = \begin{cases} q_{2\beta} - q_{2\alpha} & \text{if } s_\beta > s_\alpha \\ 0 & \text{if } s_\beta \leq s_\alpha \end{cases}, \quad s = \max\{s_\beta - s_\alpha, 0\}$$

From this we can get the Δ-intercept of the line $q_2 + s\Delta$. Let us call this p_Δ. Clearly, $p_1 \leq p_\Delta \leq p_2$ (see Fig. 8). $p_\Delta = -\frac{q_2}{s}$ if $s > 0$ and $p_\Delta = 0$ if $s = 0$. Now, if p_1 happens to be equal to p_Δ then $r = s$, otherwise r is equal to the slope of $\{\beta^l(u) - \alpha^u(u)\}$ at $u = p_1$. For example, in the case we considered before, where $p_{1\beta} \leq p_{2\beta} \leq p_\alpha$, if p_1 happens to lie between $p_{1\beta}$ and $p_{2\beta}$, then $r = r_\beta - r_\alpha$. Similarly, if p_1 lies between $p_{2\beta}$ and p_α, then $r = s_\beta - r_\alpha$. Hence q_1 is equal to $-rp_1$.

Therefore, as in the case where the arrival and the service curves were approximated by two line segments, in this case also the remaining upper and lower arrival and service curves can be efficiently approximated. This was illustrated above in the calculation of the remaining lower service curve.

4. Multiobjective Design Space Exploration

There are several possibilities for exploring the design space, one of which is a branch and bound search algorithm where the problem is specified in the form of integer linear equations (see [11]). For complicated examples where the design space can be very large, it is possible to use evolutionary search techniques (see [1]), and this is the approach we describe here.

As already mentioned, we are faced with a number of conflicting objectives trading cost against performance, and there are also conflicts arising from the different usage scenarios of the processor. We illustrate this in the case study, which involves tradeoffs between the performance ψ_b in several different usage scenarios $b \in B$ and the cost of the system architecture. Note that a usage scenario is defined by a certain set of flows F and by associated deadlines d_f. As a consequence, the binding of task to resource instances and the memory requirements may vary from scenario to scenario.

Definition 8 (Cost Measure) *The system cost is defined by the sum of costs for all allocated resource instances.*

$$cost = \sum_{s \in S} alloc(s) cost(s) \qquad (12)$$

Definition 9 (Performance Measure) *Given a system architecture as defined in Def. 7, its performance under a scenario $b \in B$ is defined as a scaling parameter ψ_b which is the largest scaling of packet input streams according to $[\psi_b \alpha_f^l, \psi_b \alpha_f^u]$ for all streams f which are part of scenario b such that the constraints on end-to-end delays and memory are satisfied. In other words, given the scenario b, we have $delay \leq d_f$ for all flows f and $\sum_{f \in F} backlog \leq m(b)$ for a given shared memory constraint $m(b)$.*

As described in Section 3.3, we may also perform a refined per-instance memory analysis for each resource so that the (possibly weighted) sum of the local memories must be less or equal the memory constraint $m(b)$.

The basic approach is shown in Fig. 9. The evolutionary multiobjective optimizer determines a feasible binding,

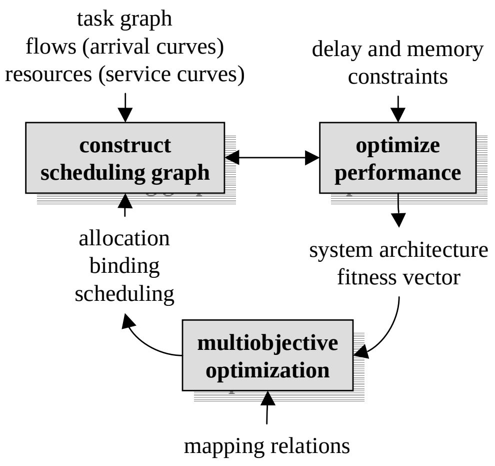

Figure 9. Basic concept for the design space exploration of packet processing systems.

allocation, and scheduling strategy based on the cost of the system architecture and the performance for each usage scenario. Based on this information, a scheduling graph is constructed for each usage scenario which enables the computation of the corresponding memory and delay properties. Then, the packet rates of the input streams are maximized until the delay and memory constraints as specified are violated. The corresponding scaling factors ψ_b of the input streams for each scenario $b \in B$ and the cost of system architecture $cost$ form the objective vector $v = (v_0, ..., v_{k-1})$, using $v_0 = cost$ and $v_i = 1/\psi_{b_i}$ for all $b_i \in B$, $i > 0$ to formulate a minimization problem. The goal is to determine implementations with *Pareto-optimal* [7] objective vectors. The architectures associated with Pareto-optimal objective vectors represent the tradeoffs in the network processor design.

Definition 10 (Pareto-optimal) *Given a set V of k-dimensional vectors $v \in \mathbb{R}^k$. A vector $v \in V$ dominates a vector $g \in V$ if for all elements $0 \leq i < k$ we have $v_i \leq g_i$ and for at least one element, say l, we have $v_l < g_l$. A vector is called Pareto-optimal if it is not dominated by any other vector in V.*

As can be seen, there are two optimization loops involved: The inner loop locally maximizes the throughput of the network processor in each scenario under the given memory and delay constraints. The outer loop performs the multiobjective design space exploration.

We have used a widely used evolutionary multiobjective optimizer SPEA2 (see [6, 7]) and incorporated some domain specific knowledge into the search process. The optimizer iteratively generates new system architectures based on the already known set. These new solutions are then

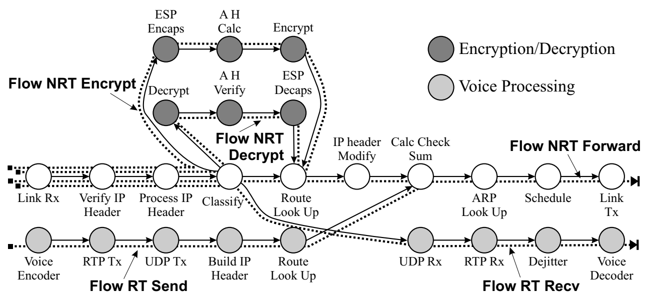

Figure 10. Task graph for a network processor.

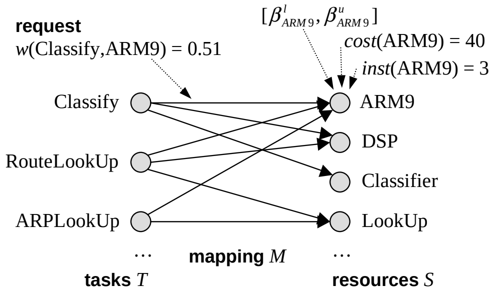

Figure 11. Graphical representation of a part of the mapping of tasks to resources.

evaluated for their objective vector. It may be noted that due to the heuristic nature of the search procedure, no statements about the optimality of the final set of solutions can be made. However, there is experimental evidence, that the found solutions are close to the optimum even for realistic problem complexities.

5. Case study

The purpose of this section is to give a complete design space exploration example. We use the same initial specification as in [16] in order to have comparable results. In particular we have the following set of traffic streams $F = \{\mathrm{NRT\_Forward}, \mathrm{RT\_Send}, \mathrm{RT\_Recv}, \mathrm{NRT\_Encrypt}, \mathrm{NRT\_Decrypt}\}$ and 25 computation tasks, i.e. $|T| = 25$. The task graph with its dependencies is visualized in Fig. 10.

Our goal is to optimize a network processor looking at two different scenarios. In the first usage scenario, we have just the stream $\mathrm{NRT\_Forward}$ to model forwarding functionality in the network backbone, whereas in the second scenario all streams in F resemble an access network environment with an increased per-packet processing requirement. We use eight different resource types with $S = \{\mathrm{Classifier}, \mathrm{PowerPC}, \mathrm{ARM9}, \mu\mathrm{Engine}, \mathrm{CheckSum}, \mathrm{Cipher}, \mathrm{DSP}, \mathrm{LookUp}\}$. Each type has different computational capabilities and these be represented in form of the mapping relation M (see Def. 6). A part of this specification is represented in Fig. 11, including an example for the implementation cost $cost(s)$, the number of instances $inst(s)$ of a resource type, and a request $w(t, s)$ of a task. There are general-purpose resources like an ARM9 CPU which is able to perform all kinds of tasks, and very specialized ones like the classifier that can only handle a single task. The initial service curves are simply set to $\beta^l(\Delta) = \beta^u(\Delta) = c \cdot \Delta$, $c \in \mathbb{R}_{>0}$, for computation and communication resources, reflecting the fact that the resources are fully available for the processing of the tasks.

It should now be obvious to the reader, why the application of an evolutionary optimizer is advantageous for our settings. Looking at the access network scenario and considering all possible bindings of our task graph in Fig. 10 to different resource types, we will have more than $4^{25} \cdot 5! > 10^{17}$ possible design points, assuming that all of the 25 tasks can at least be executed on the four general-purpose resource types (ARM9, PowerPC, μEngine, DSP) with varying requests. The factor $5!$ takes the choice of priorities for the five traffic streams into account. Note that this rough estimate does not even include the option to allocate several instances per resource type or several communication resources.

Fig. 12 shows the final population of a design space exploration run after 300 generations of a population of 100 system architectures. This optimization takes less than 30 minutes to run on a Sun Ultra 10. Most parts of the software are written in Java, including the graphical editor for the specification of tasks, resources, and bindings, and the operators in evolutionary algorithm such as mutation, crossover, and repair operators. The evolutionary optimizer is written in C++. Each dot in Figure 12 represents a Pareto-optimal system architecture. This includes (a) the set of allocated resources, (b) the binding of tasks to resource instances for each scenario, and (c) the scheduling priorities for each scenario. The three-dimensional design space is defined by the costs of resource allocations and the performance factors ψ for our two scenarios (backbone and access networks) which are bound by either the end-to-end deadlines associated with the streams or the maximum memory constraints (see Def. 9). For visualization purposes we transformed the cost values $cost$ by $cost := cost_{max} - cost$ (where $cost_{max}$ is the maximum cost value in the population) to have cheap solutions on top of the hill.

We can recognize two distinct regions of solutions. The region on the left includes solutions which are in particular good for the network backbone whereas the region in the middle shows designs that can be used for both usage scenarios. Note that there are no solutions in the region on the

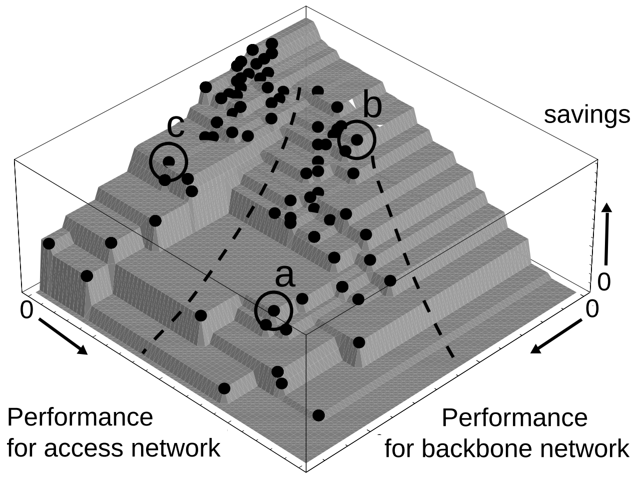

Figure 12. Final population of a design space exploration run. The three-dimensional space is defined by the performance (ψ) values for two usage scenarios and the savings in costs. All plotted designs are Pareto-optimal.

right since architectures which show good performance for access networks must also inherently perform well for the backbone, because the flows for the backbone scenario are included in the access scenario. A major characteristic of the region in the middle is that allocations contained in that area require an expensive cipher unit to cope with the diverse set of flows in the access network scenario. Solutions in the left region however do not use a cipher component.

The allocations for three selected Pareto-optimal design points are sketched in Fig. 13 to show an example of the tradeoffs involved in network processor design. A higher average utilization of a resource is denoted by a darker coloring in the figure. We bounded the exploration to a single communication instance where each allocation had a choice between two different bus types. The designs b) and c) in particular optimize the performance for one of the two usage scenarios at the same (moderate) costs, whereas design a) performs even slightly better in both scenarios at more than double the cost. Depending on the targeted application domain, each of the solutions might be meaningful. Design b) which performs well for the access network scenario requires a relatively expensive cipher unit to cope with encryption and decryption. Design c) which is aimed for IP forwarding in backbone networks, can however tradeoff the cost for an unnecessary cipher unit to allocate more general-purpose computing power at the same cost. Finally, design a) is well-suited for both scenarios and therefore requires an extensive allocation of resources which actually is a mixture of the allocations for the designs b) and c). Note that all tasks which are bound to the relatively cheap CheckSum resource in the designs b) and c) are now bound to unexploited ARM resources in design a).

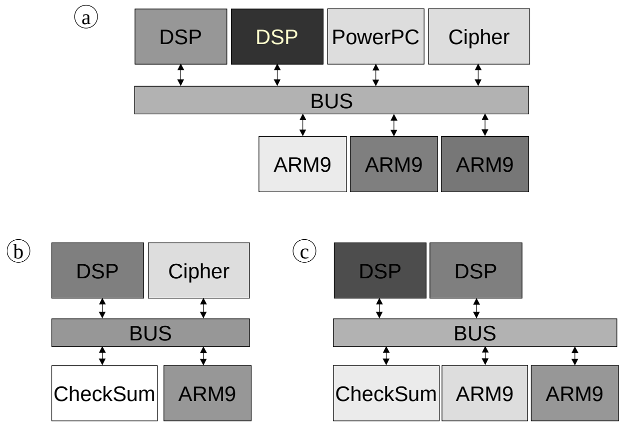

Figure 13. Examples for Pareto-optimal resource allocations taken from Fig. 12. Darker coloring means higher average utilization.

Due to space limitations we do not discuss the exact parameters for modelling our traffic streams, the costs, and the requests of tasks. For the same reason, we do not provide quantitative results for the memory consumption, but describe qualitative memory requirements. Design c) aimed for the network backbone needs most of the memory for two resource instances only. About 60% of the memory space defined by the memory constraint (see Def. 9) is given to the DSP instance with the higher average load and the remaining 40% are allocated to one ARM instance (again the one with the higher load value). Compared with that, design b) targeted to access networks reverses the memory consumption by allocating one quarter of the memory space for the DSP, two third for the ARM, and the remaining 8% for the cipher unit. For the costly design a), no simple memory consumption pattern can be recognized. If a network processor according to design a) is alternately used for both scenarios (which would be a rather unlikely case), it would be interesting to note that the memory constraint must only be increased by 17.5% to accommodate for each resource instance the maximum required memory area of both the scenarios.

The performance of the allocation and the memory consumption are determined using our analytical approach based on the scheduling network. An example is given in Fig. 14 for design b), looking at a decryption traffic stream. Besides the order in which arrival and service curves are derived, we also recognize the allocation of resources and the binding of tasks to resources in the network.

6. Future Work

We have presented a new approach towards the modelling and design space exploration of network processor

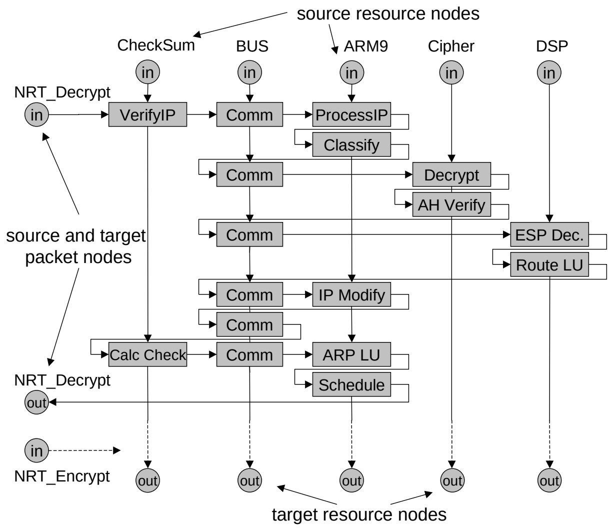

Figure 14. Scheduling network of one flow for architecture b) in Fig. 13. The scheduling policy for each resource is fixed priority. The internal scheduling nodes correspond to the basic blocks shown in Fig. 5.

architectures. Our method is based on a very high level of abstraction where the goal is to quickly identify interesting regions of the design space. There is, however, scope for taking into account several additional details which we have not considered in this paper. For example, all the memory units we have considered here are assumed to be embedded in some processing resource and are all of the same type. However, for high performance settings, such as in backbone networks, network processors use several different types of memory units such as SRAMS (for storing lookup tables, for instance), DRAMS, and also embedded memory of the kind that was considered here. Modeling this will improve the accuracy of our results.

We are also in the process of investigating the feasibility of our approach to more elaborate and realistic examples and towards this we are collaborating with IBM Research Zürich for realistic input data and traffic flows.

Acknowledgement

The work presented in this paper has been supported by IBM Research. The authors are grateful to Andreas Herkersdorf, IBM, for discussions that influenced the presented results.

References

[1] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. *Design Automation for Embedded Systems*, 3(1):23–58, 1998.

[2] J. L. Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050, Springer Verlag, 2001.

[3] G. Buttazzo. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[4] P. Crowley, M. Fiuczynski, J.-L. Baer, and B. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proc. International Conference on Supercomputing*, Santa Fe, 2000.

[5] R. Cruz. A calculus for network delay. *IEEE Trans. on Information Theory*, 37(1):114–141, 1991.

[6] K. Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley, Chichester, 2001.

[7] M. Eisenring, L. Thiele, and E. Zitzler. Handling conflicting criteria in embedded system design. *IEEE Design & Test of Computers*, 17(2):51–59, 2000.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[9] K. Lahiri, A. Raghunathan, and S. Dey. System level performance analysis for designing on-chip communication architectures. *IEEE Trans. on Computer Aided-Design of Integrated Circuits and Systems*, 20(6):768–783, 2001.

[10] K. Lahiri, A. Raghunathan, and G. Lakshminarayana. LOTTERYBUS: A new high-performance communication architecture for system-on-chip designs. In *Proc. 38th Design Automation Conference*, 2001.

[11] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill International Editions, New York, 1994.

[12] L. Peterson, S. Karlin, and K. Li. OS support for general-purpose routers. In *Proc. 7th Workshop on Hot Topics in Operating Systems*, 1999.

[13] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling computations on a software-based router. In *Proc. SIGMETRICS*, 2001.

[14] S. Shenker and J. Wroclawski. General characterization parameters for integrated service network elements. RFC 2215, IETF, Sept. 1997.

[15] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proc. Symposium on Operating Systems Principles (SOSP)*, 2001.

[16] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors – models and algorithms. In *First Workshop on Embedded Software*, Lecture Notes in Computer Science 2211, pages 416–434, Lake Tahoe, CA, USA, 2001. Springer Verlag.

[17] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[18] T. Wolf, M. Franklin, and E. Spitznagel. Design tradeoffs for embedded network processors. Technical Report WUCS-00-24, Department of Computer Science, Washington University in St. Louis, 2000.