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 of architectures of packet processing devices on the system level. Our method is specific to the application domain of network packet processors and is based on (1) models for packet processing tasks, a specification of the workload generated by traffic flows, and a description of the feasible space of architectures involving 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) an evolutionary algorithm for multi-objective design space exploration. Our method is analytical and is based on a high level of abstraction, where the goal is to quickly identify interesting architectures, which may then 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 flows of data packets, where for each flow 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 be appropriately modeled? 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 such flows of packets 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, 20] for reasoning about packet flows and their processing. The task model rep-

resents 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 flows (such as their burstiness, and long-term arrival rates), which are specified using their *arrival curves* [6] and possible deadlines within which the packets must be processed.

- 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 modeled 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. Most of the previous work on modeling, performance evaluation and design space exploration of network processors (such as [5] and [4]) relied on simulation techniques, where different architectures are simulated and evaluated using benchmark workloads. The work in [5] and [4] addresses issues related to identifying appropriate workloads and modeling frameworks to aid full system analysis and evaluation using simulation. An analytical performance model for network processors was proposed in [21] and [9]. Different network processor architectures can be evaluated on benchmark workloads using this analytical model. When the search space being explored is large, it might be too expensive to evaluate all the alternatives using simulation, or even by using performance models whose input is a benchmark workload. In contrast to these approaches, different architectures in our framework are evaluated on the basis of analytical models for both the input traffic (workload) and the performance of an architecture. Based on these models we determine bounds on the resource requirements of a processor architecture (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 [19], task scheduling [15], operation system issues [14], and packet processor architectures [10, 17]. 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.

Our previous attempts to perform system-level design space exploration of packet processing architectures have been described in [19] and [18]. In [19] the exploration is performed by an integer linear program and the estimation of the system properties is limited to very simple 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. [18] overcomes some of these shortcomings and proposes a multiobjective evolutionary algorithm for the design space exploration, over the use of integer linear programming. In this paper we extend the work presented in [18] by firstly modeling communication resources (such as buses and bus arbitration policies), and secondly allowing for local memories to be associated with processing resources. We also present here a detailed design problem and show all the tradeoffs involved.

Related work on the design space exploration of SoC communication architectures includes [11] (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

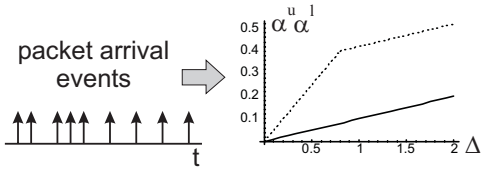

Figure 1. Representation of arrival curves.

study illustrating our methodology is presented in Section 5.

2 Models for Streams, Tasks, and Resources

In this section we describe models for the workload generated by packet flows, and task structures associated with the processing of such flows. We then make use of these models to describe our network processor architectures.

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 on 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. Such arrival curves are commonly used in the networking area for characterizing traffic flows (see, for instance, the T-SPEC model [16] of the IETF).

Let $R(t)$ denote the number of packets that have arrived from a flow f during the time interval $[0, t]$.

Definition 1 (Arrival Curves) *For any flow f, the lower arrival curve α_f^l and the upper arrival curve α_f^u, satisfy the relation:*

$$\alpha^l(t-s) \leq R(t) - R(s) \leq \alpha^u(t-s) \qquad \forall 0 \leq s \leq t$$

$\alpha_f^l(\Delta)$ gives a lower bound on the number of packets that might arrive from a flow f within any time interval of length Δ. Likewise, $\alpha_f^u(\Delta)$ gives an upper bound on the number of packets that might arrive from a flow f within any time interval of length Δ. Hence, for all $\Delta > 0$, $\alpha_f^l(\Delta) \leq \alpha_f^u(\Delta)$ and $\alpha_f^l(0) = \alpha_f^u(0) = 0$. Therefore, within any time interval of length $\Delta \in \mathbb{R}_{\geq 0}$, the number of packets arriving from a flow f is greater than or equal to $\alpha_f^l(\Delta)$, and less than or equal to $\alpha_f^u(\Delta)$.

Arrival curves may be determined from service level agreements (for example, specified using T-SPECs), by analysis of the traffic source, or by traffic measurement. Fig. 1 shows an example of an arrival curve.

All packets belonging to the same flow are processed in the same way, i.e. a fixed set of tasks are executed on each packet in a predefined order. This task structure characterizing packet processing functions can be described as follows.

Definition 2 (Task Structure) *Let F be a set of flows and T be a set of tasks. To each flow $f \in F$ there is an associated 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 flow f while respecting the precedence relations in $E(f)$.*

Tasks associated with different flows can be combined into one *conditional task graph* where depending on the flow to which a packet belongs, the packet takes different paths through this graph. See Fig. 12 for an example. Such tasks are implemented on *resources* which might be general CPUs, dedicated processors, etc.

Definition 3 (Deadlines and Requests) *To each flow $f \in F$ there is associated an end-to-end deadline d_f, denoting the maximum time by which any packet of this flow has to be processed after its arrival. If a task t can be executed on a resource s, then it creates a "request", denoting the processing requirement due to task t processing a packet on the resource s. For example, this request might represent the number of processor cycles or instructions required for processing a packet with the function described by task t. Therefore, for all possible task to resource bindings there exist a request $w(t, s) \in \mathbb{R}_{\geq 0}$.*

As defined above, the end-to-end deadline d_f denotes the maximum allowed time span from the arrival of any packet from flow f, till the end of the execution of the last task for that packet.

A network processor may be used in a variety of different usage *scenarios*, having different load conditions and delay constraints on the processing of packets. These different scenarios might lead to conflicting design objectives for the network processor. Our design space exploration scheme presented in this paper takes these conflicting goals into account and outputs all design tradeoffs (this is shown in Section 4). The set of packet flows belonging to each scenario might be different, and they might have different arrival curves. Additionally, for each scenario, there might be a different constraint on the memory available in the network processor (i.e. the maximum number of packets that might be stored in the processor at any point in time), and there might be different maximum allowable delays associated with each flow. These are formally defined in Section 4. Until then, unless otherwise mentioned, we assume that there is only one usage scenario.

2.2 System Architecture

Network processors are usually heterogeneous in nature, consisting of one or more CPU cores and dedicated processing units. Simple tasks with high data rate requirements

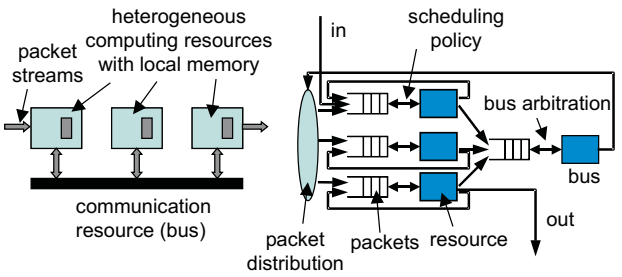

Figure 2. Example of a physical (left) and logical (right) structure of a network processor architecture.

are executed on dedicated or application specific instruction set components, whereas more complicated tasks are implemented in software running on (homogeneous) multiprocessors. In the later case, run-time scheduling methods might be used to fairly share the available resources among packets belonging to the different flows, and also to meet deadline requirements of real-time flows such as voice or video (see [3]). Each computation resource might make use of dedicated local memory such as on-chip embedded memory. If two neighboring tasks of a task graph are implemented on the same processing resource, then they do not suffer from any communication overhead. However, when such tasks are implemented on different resources, they must communicate using a communication resource (such as a bus). We would like to point out here that the task structure defined by Def. 2 may therefore also contain tasks which represent *communication tasks*. In contrast to tasks (such as header processing) where the load is defined by the number of packets, requests for communication tasks may be specified in terms of "number of bytes" involved in the transfer. Also note that 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 is described in Section 3.3. Our consideration of buses and local memories in the architecture exploration allows for more realistic representation of typical network processors, and generalizes the models presented in the previous work on this topic [18]. A sketch of a heterogeneous architecture with different packet processing paths is shown in Fig. 2 (left part of the figure).

We describe the computation or communication capabilities of resources using *service curves*. These curves are similar to the arrival curves and denote the maximum and minimum possible "service" that can be offered by a resource over any time interval of a specified length. For example, let $C(t)$ denote the number of packets that can be processed by a resource s during the time interval $[0, t]$. Then the upper and service curves β_s^u and β_s^l corresponding

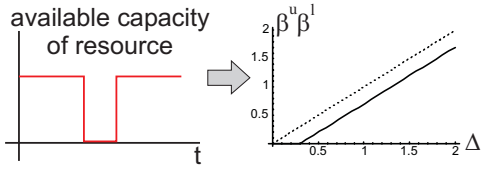

Figure 3. Representation of service curves.

to this resource satisfy the inequality:

$$\beta_s^l(t-s) \leq C(t) - C(s) \leq \beta_s^u(t-s) \qquad \forall 0 \leq s \leq t$$

and $\beta_s^l(0) = \beta_s^u(0) = 0$.

Definition 4 (Service Curves) *For any $\Delta \in \mathbb{R}_{\geq 0}$ and any resource s belonging to a set of available resources S, the lower service curve $\beta_s^l(\Delta)$ is a lower bound on the number of computing/communication units available from resource s over any time interval of length Δ. Similarly, the upper service curve $\beta_s^u(\Delta)$ denotes an upper bound on the number of computing/communication units available from resource s over any time interval of length Δ. Therefore, the computing/communication units available from resource s over any time interval of length Δ is always greater than or equal to $\beta_s^l(\Delta)$ and less than or equal to $\beta_s^u(\Delta)$.*

Clearly, if a resource is loaded with the execution of certain tasks, then the available computing power after serving these tasks will be less than the power originally available; moreover, the computing power might vary over time intervals, depending on the executing pattern of the tasks. An example of $\beta_s^l(\Delta)$ and $\beta_s^u(\Delta)$ is shown in Fig. 3.

Finally, the *set of available resources* and the task to resource mappings can be formally defined as follows.

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_s^l and an upper service curve β_s^u.*

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 $(t, s) \in M$ then task t could be executed on resource type s.*

If $(t, s) \in M$, i.e. the task t can be executed on a resource of type s, then a *request* $w(t, s) \in \mathbb{R}_{\geq 0}$ is associated with this mapping (see Def. 3).

Therefore, our model of a feasible system architecture is based on the following: (1) available resource types including their processing or communication capabilities and

performance described by service curves, (2) *costs* for implementing a resource on the network processor, for example this might be the chip area required for the resource, and finally (3) the scheduling / bus arbitration policies and associated parameters. The logical structure of a system architecture is shown in Fig. 2 (on the right hand side of the figure). Here we see that the processing components have associated memories which store packets which are waiting for the next task to be executed on them. A scheduling policy associated with the processing component selects a packet and starts the execution. The processing of the current packet may be preempted in favor of a task for processing another packet. After the execution of a task, a packet may be reinserted into the input queue of the current resource, to be processed by the next task which also executes on the same resource. Alternatively, the packet may be redistributed to another resource using a bus. Without restricting the applicability of our approach, we limit the description of suitable architectures to a single bus in order to simplify the explanation.

3 Analysis using a Scheduling Network

Although we have chosen a particularly simple cost model, it is not obvious how to determine for any resource, the maximum number of stored packets in it waiting to be processed at any point in time. Neither is it clear how to determine the maximum end-to-end delays experienced by the packets, since all packet flows share common resources. 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_s^l and β_s^u, 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, making the computations of the packet delays and the memory requirements non-trivial. Interestingly, we show that there exists a computationally efficient method to derive provably correct bounds on the end-to-end delays of packets and the required memory for each computation and communication.

We exploit the fact that characteristic chains of tasks are executed for each packet flow and that all the flows are processed independently of each other. Based on this knowledge we construct a *scheduling network*, where the *real-time calculus* (based on arrival and service curves) 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 domains (such as digital signal processing).

The basis for the determination of end-to-end delays and memory requirements is the description of packet flows in communication networks using a network calculus (see [6] for an introduction). Recently, this approach has been reformulated in an algebraic setting in [2]. In [20], a comparable approach has been used to describe the behavior of processing resources.

3.1 Building Blocks of the Scheduling Network

The basic idea behind our performance estimation (such as end-to-end delays experienced by packets when processed by a given architecture, and memory requirements of the architecture assuming a set of input traffic flows) is the provision for a network theoretic view of the system architecture. More precisely, packet flows and resource streams flow through a network of processing and communication resource nodes and thereby adapt their (output) arrival curves (of the packet flows) and (remaining) service curves (of the resources) respectively. Inputs to a network node are the arrival curves of packet flows and the service curves of the corresponding resource that the node is representing. The outputs describe the resulting arrival curves of the processed packet flows 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. As an example, see Figure 16, which is explained in details later.

In order to understand the basic concept, let us first describe a very simple example of such a node, namely the preemptive processing of packets from one flow 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 [19], the following theorem has been derived which describes the processing of a packet flow in terms of the already defined arrival and service curves.

Theorem 1 *Given a packet flow 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 arrival curve of the processed packet flow and the remaining service of the resource node. $\alpha^{l'}$ and $\alpha^{u'}$ denotes the lower and the upper arrival curve respectively of the processed flow, and $\beta^{l'}$ and $\beta^{u'}$ denotes the lower and upper remaining service curve respectively of the resource node.*

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

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

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

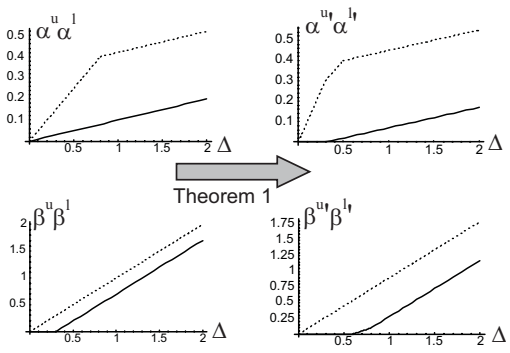

Figure 4. Remaining service and resulting arrival curve according to Theorem 1.

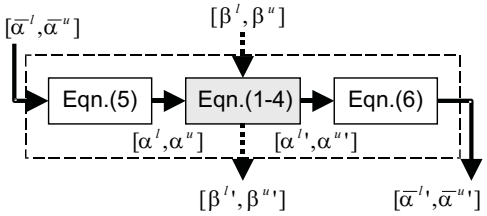

Figure 5. Block diagram showing the transformation of packet flows and resource streams by a processing device. The dotted arrows represent the resource flow, the others show the flow of packets (or the corresponding requests).

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

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 flows in the system architecture, we need to convert packets to their corresponding computing requests. Given bounds on a packet flows of the form $[\overline{\alpha}^l, \overline{\alpha}^u]$ we can determine bounds on the related computing requests

$$[\alpha^l, \alpha^u] = [w\overline{\alpha}^l, w\overline{\alpha}^u] \tag{5}$$

considering the request w for each packet (representing the processing requirement for a packet). The notation $[\alpha^l, \alpha^u]$ represents the fact that α^l and α^u are lower and upper curves of the same flow.

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

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

The whole transformation is depicted in Fig. 5.

3.2 Scheduling Policies

In this section we describe how to extend the calculation of the resulting processed arrival curves (for packet flows) and the remaining service curves (for resource flows), to the case involving multiple flows and resources. As packets from several flows arrive at a resource, they are served in an order determined by the scheduling policy. The resulting arrival curves and remaining service curves are dependent on this scheduling policy. First we give the results for the preemptive version of *Fixed Priority Scheduling*, and then for the *Generalized Processor Sharing* scheduling.

3.2.1 Fixed Priority Scheduling

For the fixed priority scheme, let us assume that there is a set of flows $f_1, \ldots, f_n$, and a resource s which serves these flows in the order of decreasing priority. Flow f_1 has the highest priority and flow f_n the lowest. For each packet of the flow f_i, a task t_i must be executed on the resource s and this creates a demand (or request) equal to $w(t_i, s)$ on s. We denote $w(t_i, s)$ by w_i in short. With each flow f_i is associated its upper and lower (packet) arrival curves $\overline{\alpha}_i^u$ and $\overline{\alpha}_i^l$. For the resource s, the flow f_i is served using the upper and lower service curves β_i^u and β_i^l respectively. The resource s in its unloaded state is described by the service curves β^u and β^l.

Because of the fixed priority scheme, the resource s serves the flows in the order of decreasing priority, and the resulting arrival curves of the flows and the remaining service curve of the resource is computed according to Theorem 1. In order to have compatible units, we need first to multiply the arrival curves with the demand for each task, namely w_i. Correspondingly, the flow leaving the resource must be divided by w_i. If the subsequent units which use the outgoing packets flowing out of s can start processing only after the whole task has been finished, then we need to apply the floor/ceiling-function to the outgoing flows (depending on whether it is the lower or the upper curve). Therefore, the outgoing arrival curves are transformed according to $\overline{\alpha}_i^{u\prime}(\Delta) = \lceil \alpha_i^{u\prime}(\Delta)/w_i \rceil$ and $\overline{\alpha}_i^{l\prime}(\Delta) = \lfloor \alpha_i^{l\prime}(\Delta)/w_i \rfloor$. The curves obtained provide the correct bounds as one can show that $\lfloor a \rfloor - \lfloor b \rfloor \le \lfloor a - b \rfloor$ and $\lceil a \rceil - \lceil b \rceil \ge \lceil a - b \rceil$. In addition to the relations shown in Theorem 1 that have to be applied for all indices $1 \le i \le n$, we have the following additional equations describing how to obtain β_i^u, β_i^l $(1 < i \le n)$ from β_1^u and β_1^l as a result of the fixed priority scheduling. These equations along with the equations for scaling as described above can be given as follows.

$$\alpha_i^u(\Delta) = w_i \cdot \overline{\alpha}_i^u(\Delta) \quad , \quad \alpha_i^l(\Delta) = w_i \cdot \overline{\alpha}_i^l(\Delta)$$

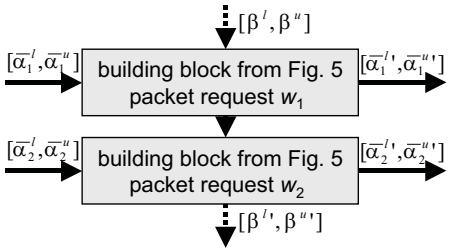

Figure 6. Representation of fixed priority preemptive scheduling of packet flows on a single processing resource. Flow 2 has a smaller priority than flow 1.

$$\overline{\alpha}_i^{u\prime}(\Delta) = \lceil \alpha_i^u(\Delta)/w_i \rceil \quad , \quad \overline{\alpha}_i^{l\prime}(\Delta) = \lfloor \alpha_i^l(\Delta)/w_i \rfloor$$

$$\beta_1^u(\Delta) = \beta^u(\Delta) \quad , \quad \beta_i^u(\Delta) = \beta_{i-1}^{u\,\prime}(\Delta) \quad \forall 1 < i \leq n$$
$$\beta^{u\prime}(\Delta) = \beta_n^{u\prime}(\Delta)$$
$$\beta_1^l(\Delta) = \beta^l(\Delta) \quad , \quad \beta_i^l(\Delta) = \beta_{i-1}^{l\,\prime}(\Delta) \quad \forall 1 < i \leq n$$
$$\beta^{l\prime}(\Delta) = \beta_n^{l\,\prime}(\Delta)$$

Figure 6 shows an example of fixed priority scheduling using two flows and one resource. Finally, note that the remaining service curves β' from the resource s (after it has processed the flows $f_1, \ldots, f_n$) can be used to service other flows, using possibly a different scheduling scheme, in an hierarchical manner. The processed flows with the resulting (packet) arrival curves $\overline{\alpha}_i^{u\prime}$ and $\overline{\alpha}_i^{l\prime}$ can now enter other resource nodes which are responsible for executing other tasks $t \in T$ or for performing communication tasks. In a later section we show how these results can be used to estimate the delay experienced by packets and the memory requirements of the resource s to hold waiting packets.

3.2.2 Generalized Processor Sharing (GPS)

As a second example we consider proportional share scheduling (GPS) [13]. In this case, with each flow f_i $(1 \leq i \leq n)$ there is an associated weight ϕ_i with $\sum_{1 \leq i \leq n} \phi_i = 1$. A flow f_i receives a share $\phi_i / \sum_{j \in J(t)} \phi_j$ of the total available service given by β from the resource node. $J(t)$ is the set of indices of flows which are backlogged at time t. Here as well each flow is processed according to the model described in Theorem 1 and illustrated in Figure 5.

If the arrival curves associated with a flow f_i are given by $\overline{\alpha}_i^u$ and $\overline{\alpha}_i^l$, and a task t_i executing on packets of this flow makes a demand w_i on the resource, then we have the following scaled arrival curves as before.

$$\alpha_i^u(\Delta) = w_i \cdot \overline{\alpha}_i^u(\Delta) \quad , \quad \alpha_i^l(\Delta) = w_i \cdot \overline{\alpha}_i^l(\Delta)$$

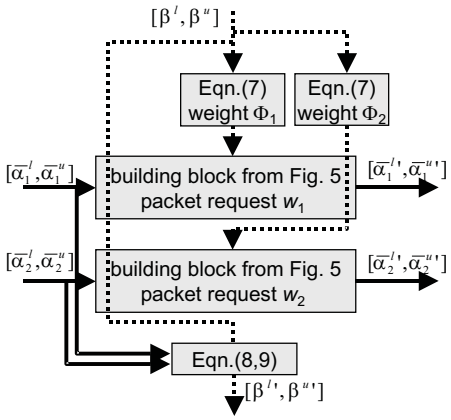

Figure 7. Deriving the resulting arrival curves and remaining service curves under the generalized processor sharing (GPS) scheduling discipline. Here two flows are scheduled on a single processing/communication resource

$$\overline{\alpha}_i^{u\prime}(\Delta) = \lceil \alpha_i^u(\Delta)/w_i \rceil \quad , \quad \overline{\alpha}_i^{l\prime}(\Delta) = \lfloor \alpha_i^l(\Delta)/w_i \rfloor$$

Since a flow f_i is served using a service curve proportional to ϕ_i of the original service curve corresponding to the resource, we can obtain the following bounds.

$$\beta_i^l(\Delta) = \phi_i \cdot \beta^l(\Delta) \quad , \quad \beta_i^u(\Delta) = \beta^u(\Delta) \qquad \forall 1 \leq i \leq n \tag{7}$$

The remaining service curve after processing the packet flows can be given in accordance with Theorem 1.

$$\beta^{l\prime}(\Delta) = \sup_{0 \leq u \leq \Delta} \{\beta^l(u) - \sum_{1 \leq i \leq n} \alpha_i^u(u)\} \tag{8}$$

$$\beta^{u\prime}(\Delta) = \sup_{0 \leq u \leq \Delta} \{\beta^u(u) - \sum_{1 \leq i \leq n} \alpha_i^l(u)\} \tag{9}$$

Figure 7 shows an example with two flows. The case of non-preemptive scheduling can be handled by shifting the service curves $\beta_i^l(\Delta)$ appropriately, i.e. by replacing them with $\beta_i^l(\Delta - w_{\max})$ when $\Delta \geq w_{\max}$ and by $\beta_i^l(\Delta)$ when $\Delta \leq w_{\max}$ where $w_{\max} = \max\{w_i : 1 \leq i \leq n\}$. Here $w_{\max}$ might represent the processing time of a packet.

Similar techniques can be used to describe other scheduling algorithms such as First-Come-First-Served or Earliest Deadline First. However, as pointed out in [2], the problem of determining accurate bounds for these scheduling disciplines is still an area of research.

3.3 Scheduling Network Construction

The scheduling network which enables the computation of performance parameters such as the end-to-end delays of

packets and the memory requirements of the network processor, is constructed using the data from the specifications given in Section 2. These consist of a specification of the input packet flows, the processing functions associated with each flow (Section 2.1), and a specification of the system architecture, such as the types of the different processing and communication resources available (Section 2.2). 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 parameters describing the packet flows (i.e. the upper and lower arrival curves) pass from one resource to the next, and in the process get modified. Here 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 parameters describing the *resource flows* (i.e. the capabilities of the resources described by the upper and lower service curves) also pass through the network and in the process get modified. The order here is determined by the priorities assigned to the packet flows (in the case of fixed priority scheduling) and the precedence relations in the task graph.

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 a task $t \in T$ to a resource 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 fixed priority 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 described not only by the type and the number of resource components, but also by the binding of the tasks to these components. This mapping may depend on the flow in which the task is active and on the scenario (see Section 2.1) under which the system architecture is evaluated.

In the target architecture, on the one hand it is possible to have dedicated hardware modules for certain tasks with the resulting architecture being heterogeneous. On the other hand, we may have parallel resource instances of the same type (i.e. $alloc(s) > 1$) which may process a complete packet flow.

Now, we can describe the construction of a scheduling network for a given scenario. Note that in general we will have different scheduling networks for each usage scenario as the tasks, flows, and priorities might 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 flow f present in the scenario.

- Construct an ordered set of tuples T_f which contains (t, f) for all flows f in the scenario and for all tasks $t \in V(f)$ in this flow. Order these tuples according to the priorities of the corresponding flows 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 flows 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 flow output of scheduling node (t_i, f) is connected to the packet flow 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) precedes (t_j, f) in the ordered set T_f, then connect the resource stream output of (t_i, f) to the resource stream 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 flows and resource streams, as well as scheduling nodes which represent the computations described in Fig. 5. An example scheduling network is given in Fig. 16.

Given the arrival curves for all the packet flows in the source nodes (i.e. $[\alpha_f^l, \alpha_f^u]$ for all flows 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 in terms of their arrival and service curves. Now, it only remains to be seen how we can determine the end-to-end delays of the packets and the necessary memory required to hold packets waiting to be served.

3.4 System Properties

In order to estimate the properties of a system architecture for a network processor, we need bounds on the end-to-end delays experienced by the packets being processed and bounds on the memory requirements of the processor. Using well known results from the area of communication networks (see e.g. [6]), the bounds derived in Theorem 1 can be used to determine the maximum delays of the packets and the necessary memory required to store waiting packets.

$$delay \leq \sup_{u \geq 0} \left\{ \inf\{\tau \geq 0 : \alpha^u(u) \leq \beta^l(u+\tau)\} \right\} \quad (10)$$

$$backlog \leq \sup_{u \geq 0} \{\alpha^u(u) - \beta^l(u)\} \quad (11)$$

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 the case of the scheduling network constructed above, we need to know which curves to use in the inequalities (10) and (11). The upper arrival curve is that of an incoming packet flow, i.e. α_f^u of the flow f being investigated in the current scenario. The service curve β^l to be used in the inequalities (10) and (11) is the *accumulated* curve of all scheduling nodes through which the packets of flow f pass in the current scenario. As described in e.g. [2], this quantity can be determined through an iterated convolution. To this end, let us suppose that the packets of flow f pass through scheduling nodes $p_1, ..., p_m$ which have the lower service curves $\beta_1^l, ..., \beta_m^l$ at their resource stream inputs. Then β^l in (10) and (11) can be determined using the following recursion:

$$\overline{\beta}_1^l = \beta_1^l \quad (12)$$

$$\overline{\beta}_{i+1}^l = \inf_{0 \leq u \leq \Delta} \left\{ \overline{\beta}_i^l(u) + \beta_{i+1}^l(\Delta - u) \right\} \forall i > 1 \quad (13)$$

$$\beta^l = \overline{\beta}_{m+1}^l \quad (14)$$

As a result, using the scheduling network described above we can compute bounds on $delay$ which gives the maximum delay experienced by packets of a flow f, and the maximum shared memory $backlog$ required by the flow. If we are interested in the memory requirements for an implementation with separate local memories as shown in Fig. 2, we can generate the accumulated service curves for all sequences of tasks which are implemented on the same resource instance. There are several special cases, where we can make use of an accumulated service curve to determine tighter bounds, compared to independently deriving memory requirements for each node. For instance, suppose that a packet flow is processed first on a general-purpose component. For a certain task the flow is then delegated to a more specialized unit. After being processed on that dedicated resource instance, the flow 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 flow at that resource. We will not describe all the subcases here since the form of the equations (10) to (14) is not affected.

The memory requirements derived by an analysis of the 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.

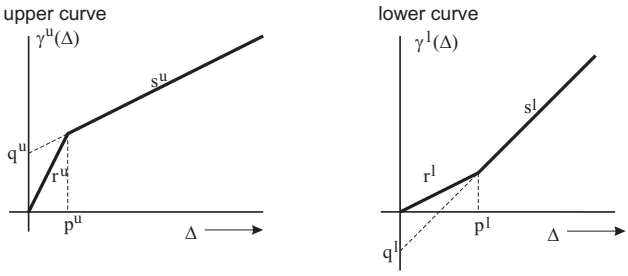

Figure 8. Simple representation of upper and lower curves.

3.5 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 schemes like evolutionary multiobjective algorithms (see [7]), the performance of many candidate 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. We only describe the basic concepts here and give a few simple examples. Fig. 8 shows how the arrival and service curves look like when each curve is approximated by a combination of two line segments.
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,

$$q^u \geq 0, \quad r^u \geq s^u \geq 0, \quad r^u = s^u \Leftrightarrow q^u = 0$$
$$q^l \leq 0, \quad 0 \leq r^l \leq s^l, \quad r^l = s^l \Leftrightarrow q^l = 0$$

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

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

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

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

Note that $\beta^l(\Delta) - \alpha^u(\Delta)$ and also $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ 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

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

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 the 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.

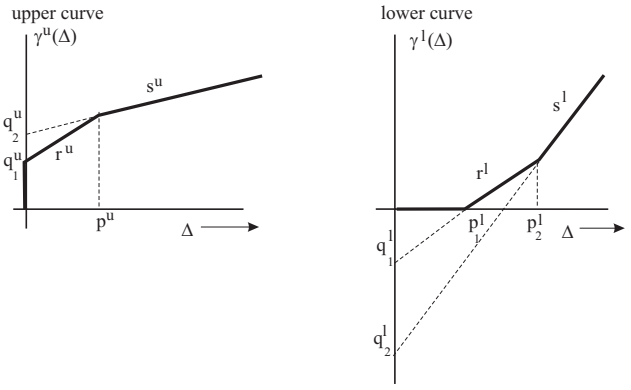

Figure 9. Improved approximation of upper and lower curves.

3.6 Improved Approximations

In this section we show that it is possible to obtain improved approximations of the remaining arrival and service curves, by approximating these curves using three line segments instead of two as in Section 3.5. The resulting calculations however become more involved in this case. Figure 9 shows the resulting arrival and service curves. This allows us to exactly model an arrival curve in the form of a T-SPEC [16]. In the case of an arrival curve, q_1^u may represent the maximum possible workload involved in processing a single packet, r^u can be interpreted as the burst rate and s^u the long term arrival rate. In the case of communication resources, q_1^u represents the maximum size of a packet.

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

$$\gamma^u(\Delta) = \min\{q_1^u + r^u\Delta, q_2^u + s^u\Delta\}$$
$$\gamma^l(\Delta) = \max\{q_2^l + s^l\Delta, q_1^l + r^l\Delta, 0\}$$

where,

$$q_2^u \geq q_1^u \geq 0, \quad r^u \geq s^u \geq 0, \quad r^u = s^u \Leftrightarrow q_1^u = q_2^u$$
$$q_2^l \leq q_1^l \leq 0, \quad 0 \leq r^l \leq s^l, \quad r^l = s^l \Leftrightarrow q_1^l = q_2^l$$

The values of p^u and p_1^l, p_2^l (see Fig. 9) 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.

Theorem 3 *Given the upper arrival and lower service curves $\alpha^u = U(q_{1\alpha}, q_{2\alpha}, r_\alpha, s_\alpha)$ and $\beta^l =$*

$L(q_{1\beta}, q_{2\beta}, r_\beta, s_\beta)$ *respectively, the approximate remaining service curve* $\beta^{l'} = L(q_1, q_2, r, s)$ *can be given by the following four cases.*

1: *There exists a* $\Delta' > 0$, *such that* $q_{2\beta} + s_\beta \Delta' = q_{2\alpha} + s_\alpha \Delta'$, *and for all* $\Delta < \Delta'$, $\alpha^u(\Delta) > \beta^l(\Delta)$. *In this case,* $r = 0, s = s_\beta - s_\alpha$ *and* $q_1 = 0, q_2 = q_{2\beta} - q_{2\alpha}$

2: *There exists a* $\Delta' > 0$, *such that* $q_{1\beta} + r_\beta \Delta' = q_{2\alpha} + s_\alpha \Delta'$, *and for all* $\Delta < \Delta'$, $\alpha^u(\Delta) > \beta^l(\Delta)$. *In this case,* $r = r_\beta - s_\alpha, s = s_\beta - s_\alpha$ *and* $q_1 = q_{1\beta} - q_{2\alpha}, q_2 = q_{2\beta} - q_{2\alpha}$

3: *There exists a* $\Delta' > 0$, *such that* $q_{1\beta} + r_\beta \Delta' = q_{1\alpha} + r_\alpha \Delta'$, *and for all* $\Delta < \Delta'$, $\alpha^u(\Delta) > \beta^l(\Delta)$. *In this case,* $r = r_\beta - r_\alpha, s = s_\beta - s_\alpha$ *and* $q_1 = q_{1\beta} - q_{1\alpha}, q_2 = q_{2\beta} - q_{2\alpha}$

4: *There exists a* $\Delta' > 0$, *such that* $q_{2\beta} + s_\beta \Delta' = q_{1\alpha} + r_\alpha \Delta'$, *and for all* $\Delta < \Delta'$, $\alpha^u(\Delta) > \beta^l(\Delta)$. *In this case,* $r = s_\beta - r_\alpha, s = s_\beta - s_\alpha$ *and* $q_1 = q_{2\beta} - q_{1\alpha}, q_2 = q_{2\beta} - q_{2\alpha}$

If $\alpha^u(\Delta) \geq \beta^l(\Delta)$ *for all* $\Delta \geq 0$ *then* $r = s = 0$ *and* $q_1 = q_2 = 0$

Proof. To prove that $\beta^{l'} = L(q_1, q_2, r, s)$ is a valid lower remaining service curve, we shall as before show that $L(q_1, q_2, r, s)(\Delta) \leq \sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ for all $\Delta \geq 0$.

Firstly, it may be noted that $\beta^l(\Delta)$ and $\alpha^u(\Delta)$ are convex and concave respectively. Therefore, $\beta^l(\Delta) - \alpha^u(\Delta)$ and $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ are convex. However, in contrast to our approximations with two segments in Section 3.5, here we have to consider four different cases.

Case 1 is when the last segment of $\beta^l(\Delta)$ intersects the last segment of $\alpha^u(\Delta)$, at say $\Delta = \Delta'$ (see Figure 10(a)). Therefore, for all $\Delta < \Delta'$, $\beta^l(\Delta) < \alpha^u(\Delta)$. Hence, $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\} \leq 0$ for all $\Delta \leq \Delta'$, and therefore $r = 0$ and $q_1 = 0$. When $\Delta \to \infty$, $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\} = \sup_{0 \leq u \leq \Delta}\{q_{2\beta} + s_\beta u - q_{2\alpha} - s_\alpha u\}$. Therefore, we have $s = s_\beta - s_\alpha$ and $q_2 = q_{2\beta} - q_{2\alpha}$.

Case 2 is when the middle segment of $\beta^l(\Delta)$ intersects the last segment of $\alpha^u(\Delta)$. If this intersection is at $\Delta = \Delta'$, then for all $\Delta < \Delta'$, $\beta^l(\Delta) < \alpha^u(\Delta)$ and $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\} \leq 0$ for all $\Delta \leq \Delta'$. This case is shown in Figure 10(b). Clearly, $r = r_\beta - s_\alpha$, $s = s_\beta - s_\alpha$, $q_1 = q_{1\beta} - q_{2\alpha}$ and $q_2 = q_{2\beta} - q_{2\alpha}$.

Case 3 is when the middle segment of $\beta^l(\Delta)$ intersects the middle segment of $\alpha^u(\Delta)$ (see Figure 10(c)). In this case, $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ is made up of four linear segments. But we approximate it using $L(q_1, q_2, r, s)(\Delta)$, which is made up of three segments. There can be two possible subcases, the first is when $p_{2\beta} \geq p_\alpha$ (as shown in Figure 10(c)), and the second is when $p_{2\beta} < p_\alpha$. If

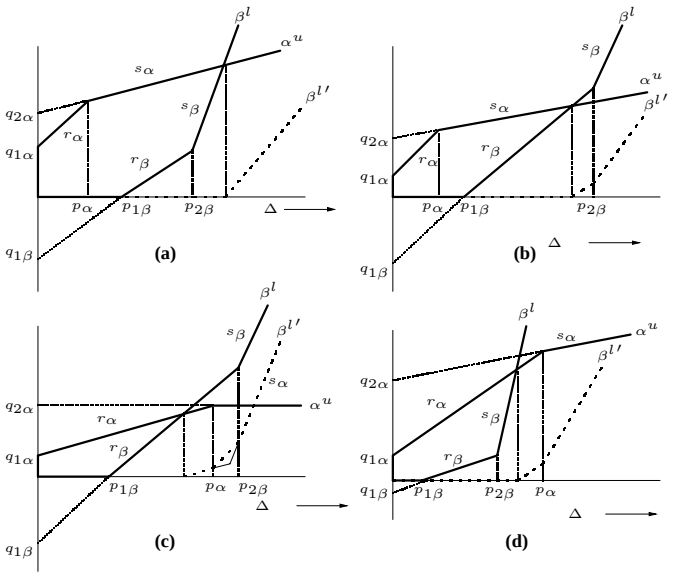

Figure 10. Approximate remaining lower service curves. Figures (a), (b), (c) and (d) represent the cases 1, 2, 3 and 4 respectively in Theorem 3

$\beta^l(\Delta)$ and $\alpha^u(\Delta)$ intersect at Δ', then the four segments that make up $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ span the intervals $\Delta \in [0, \Delta')$, $[\Delta', p_\alpha)$, $[p_\alpha, p_{2\beta})$, $[p_{2\beta}, \infty)$ (as shown in Figure 10(c)) or $\Delta \in [0, \Delta')$, $[\Delta', p_{2\beta})$, $[p_{2\beta}, p_\alpha)$, $[p_\alpha, \infty)$ (in the case when $p_{2\beta} < p_\alpha$). To obtain $L(q_1, q_2, r, s)(\Delta)$, we neglect the segment of $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ corresponding to the interval $[p_\alpha, p_{2\beta})$ in Figure 10(c) and the interval $[p_{2\beta}, p_\alpha)$ when $p_{2\beta} < p_\alpha$, and instead approximate this segment by the segments preceding and following it. It may be noted that $L(q_1, q_2, r, s)(\Delta)$ is a valid lower curve, since $L(q_1, q_2, r, s)(\Delta) \leq \sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\}$ for all $\Delta \geq 0$. Therefore, $r = r_\beta - r_\alpha$, $s = s_\beta - s_\alpha$, $q_1 = q_{1\beta} - q_{1\alpha}$, $q_2 = q_{2\beta} - q_{2\alpha}$.

Case 4 is when the last segment of $\beta^l(\Delta)$ intersects the middle segment of $\alpha^u(\Delta)$ (see Figure 10(d)). It can be seen that $r = s_\beta - r_\alpha$, $q_1 = q_{2\beta} - q_{1\alpha}$, and as before, $s = s_\beta - s_\alpha$ and $q_2 = q_{2\beta} - q_{2\alpha}$.

Lastly, if $\beta^l(\Delta) \leq \alpha^u(\Delta)$ for all $\Delta \geq 0$, then $\sup_{0 \leq u \leq \Delta}\{\beta^l(u) - \alpha^u(u)\} \leq 0$ for all $\Delta \geq 0$. Hence, $r = s = 0$ and $q_1 = q_2 = 0$. □

The approximations for all the remaining curves can be derived on the basis of similar techniques, and hence we omit them here.

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 [12]). 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. Recall from Section 2.1 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.*

$$cost = \sum_{s \in S} alloc(s) cost(s) \tag{15}$$

Definition 9 (Performance Measure) *Given a system architecture as defined in Def. 7, its performance under a scenario $b \in B$ is defined by a scaling parameter ψ_b which is the largest possible scaling of packet input flows according to $[\psi_b \alpha_f^l, \psi_b \alpha_f^u]$ for all flows f which are part of scenario b such that the constraints on end-to-end delays and memory can still be satisfied. In other words, given the scenario b, after the scaling we still have 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 3.4, 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 to the memory constraint $m(b)$.

The basic approach is shown in Fig. 11. The evolutionary multiobjective optimizer determines a feasible binding, 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 $v = (v_0, ..., v_{k-1})$,

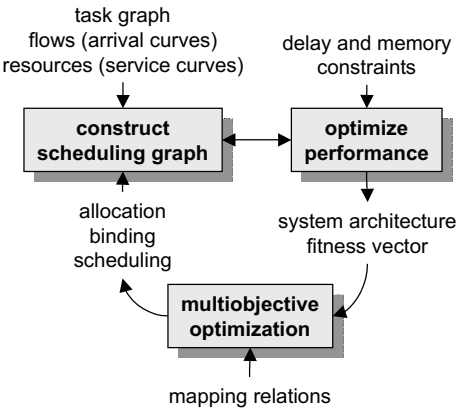

Figure 11. Basic concept for the design space exploration of packet processing systems.

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* [8] 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 [7, 8]) 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 evaluated for their objective vector and fed into the optimizer to guide further search. 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 solutions found are close to the optimum even for realistic problem complexities.

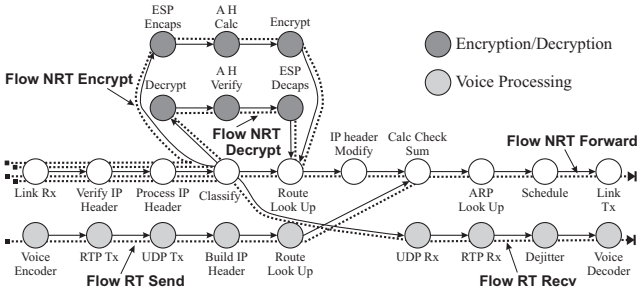

Figure 12. Task graph for a network processor.

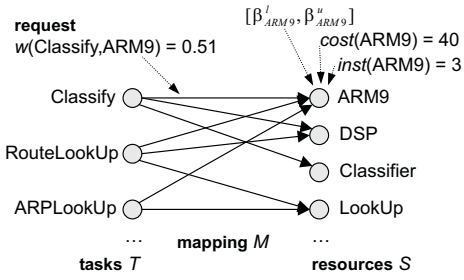

Figure 13. Graphical representation of a part of the mapping of tasks to resources.

5 Case study

The purpose of this section is to give a complete design space exploration example. For the specification, we use the following set of traffic flows $F = \{\mathrm{NRT\_Forward}, \mathrm{RT\_Send}, \mathrm{RT\_Recv}, \mathrm{NRT\_Encrypt}, \mathrm{NRT\_Decrypt}\}$ and there are 25 computation tasks, i.e. $|T| = 25$. The task graph with its dependencies is visualized in Fig. 12.

Our goal is to optimize a network processor looking at two different scenarios. In the first usage scenario, we have just the flow $\mathrm{NRT\_Forward}$ to model a forwarding functionality in the network backbone, whereas in the second scenario all the flows in F are present, representing 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 are represented in the form of the mapping relation M (see Def. 6). A part of this specification is represented in Fig. 13, 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 setting. Looking at the access network scenario and considering all possible bindings of our task graph in Fig. 12 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.

Using an evolutionary optimization algorithm, Fig. 14 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 our 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 14 represents a pareto-optimal system architecture. Each architecture 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.

In this example 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 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 also included in the access network scenario. A major characteristic of the region in the middle is that allocations contained in this 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.

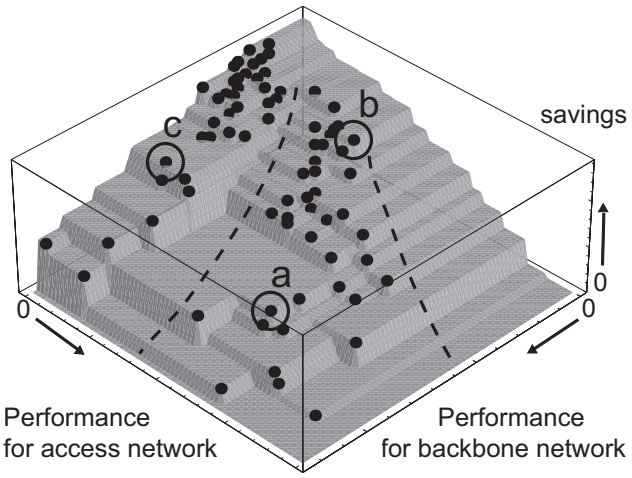

Figure 14. Final population of a design space exploration run. The three-dimensional space is defined by the performance (ψ) values for two usage scenarios and the savings in costs. All plotted designs are Pareto-optimal.

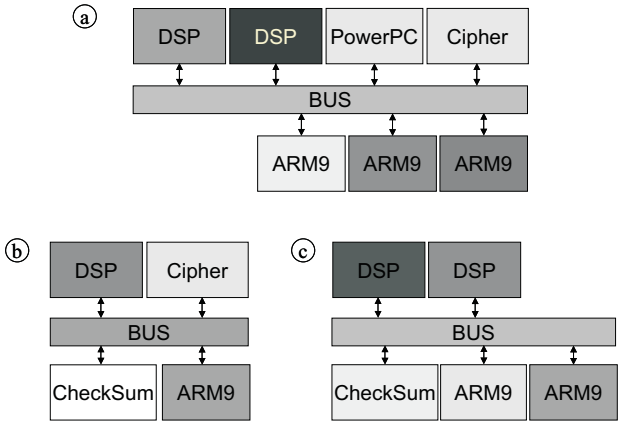

Figure 15. Examples for Pareto-optimal resource allocations taken from Fig. 14. Darker coloring means higher average utilization.

The allocations for three selected pareto-optimal design points are sketched in Fig. 15 to show an example of the tradeoffs involved in a 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).

Due to space limitations we do not discuss the exact parameters for modeling 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 thirds for the ARM, and the remaining 8% for the cipher unit. For the expensive 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. 16 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 modeling and design space exploration of network processor 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

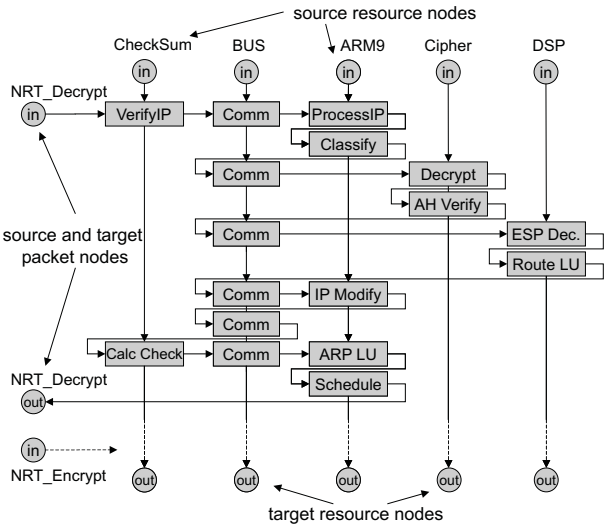

Figure 16. Scheduling network of one flow for architecture b) in Fig. 15. The scheduling policy for each resource is fixed priority. The internal scheduling nodes correspond to the basic blocks shown in Fig. 5.

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.

Acknowledgment

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

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 and J.-L. Baer. A modeling framework for network processor systems. In *Proc. 1st Workshop on Network Processors, held in conjunction with the 8th International Symposium on High-Performance Computer Architecture*, Cambridge, Massachusetts, February 2002.

[5] 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.

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

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

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

[9] M. Franklin and T. Wolf. A network processor performance and design model with benchmark parameterization. In *Proc. 1st Workshop on Network Processors, held in conjunction with the 8th International Symposium on High-Performance Computer Architecture*, Cambridge, Massachusetts, February 2002.

[10] 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.

[11] 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.

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

[13] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

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

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

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

[17] 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.

[18] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proc. 39th Design Automation Conference (DAC)*, New Orleans, USA, 2002. ACM Press.

[19] 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.

[20] 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.

[21] 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.