A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures

Lothar Thiele, Samarjit Chakraborty, Matthias Gries, Simon Künzli
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zürich
CH-8092 Zürich, Switzerland
{thiele, samarjit, gries, kuenzli}@tik.ee.ethz.ch

ABSTRACT

We present an analytical method to evaluate embedded network packet processor architectures, and to explore their design space. Our approach is in contrast to those based on simulation, which tend to be infeasible when the design space is very large. We illustrate the feasibility of our method using a detailed case study.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Modeling of computer architecture, System architectures*; B.4.1 [**Hardware**]: Input/output and data communications—*Data communication devices*

General Terms

Performance, Design

1. INTRODUCTION

Architectures for embedded packet processing devices like network processors consist of a heterogeneous combination of different hardware and software components. These include microprocessors, domain specific instruction set processors, reconfigurable hardware modules, IP cores, application specific units, communication primitives, and memory modules. A high-level design space exploration of such packet processors is concerned with resource allocation and partitioning, mapping of different packet processing functions to the different hardware and software units, and determining a scheduling policy at each of these units. These must take into account issues such as cost, power consumption, memory requirements, delays experienced by the packets that are to be processed and their possible throughputs.

There are many issues that are specific to such a design space exploration in the domain of network packet processors–heterogeneity of the processing elements that constitute the target architecture, QoS guarantees that the processed real-time traffic streams (such as voice and video) must meet, different application scenarios where such a processor might be deployed. For example, a network processor to be used in a backbone network can be characterized by very high throughput demands but relatively low processing requirements per packet, whereas access networks have lower throughput demands but high computational requirements. Due to these differences with other target domains for system-level design space exploration, several questions arise. These involve appropriate modeling of packet streams, packet processing tasks, and hardware and software resources. Other issues include performance modeling in the case of several (possibly conflicting) usage scenarios, and strategies to efficiently explore large design spaces and to obtain a reasonable compromise between multiple conflicting criteria.

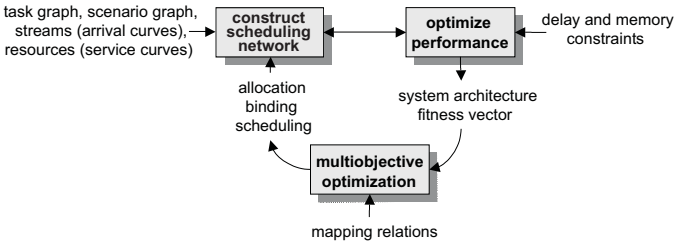

Figure 1: Basic concept for the design space exploration of packet processing systems.

In an attempt to address these issues, we present a framework which can be used to analytically evaluate packet processing architectures and identify the different design tradeoffs involved. When interfaced with a search strategy this gives an efficient means for exploring large design spaces. The search strategy comes up with possible alternatives from the design space, which are evaluated using our framework and the feedback guides further search.

The essential ingredients of our framework are a task and a resource model, and a *real-time calculus* [2, 12] for reasoning about packet streams and their processing. The task model represents 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 costs associated with each of these mappings. Traffic streams are specified using their *arrival curves* [5] and deadlines are associated with the real-time traffic. Given any architecture, we can now analytically determine properties such as delay and throughput experienced by the different traffic streams, taking into account the underlying scheduling disciplines at each of the resources.

The overall scheme used to illustrate the feasibility of our approach for design space exploration is shown in Fig. 1. Here the search strategy is a multiobjective optimizer (such as an evolutionary algorithm [6]) which comes up with possible architectures including allocation, binding, and scheduling information. Using this information, a *scheduling network* is constructed (see Section 3.2) for each usage scenario of the packet processor, which enables the computation of the memory requirements and delay properties of each traffic stream. Then the packet rates for each of the input streams are maximized until one of the delay constraints associated with the streams, or a memory constraint associated with the archi-

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

tecture is violated (see Section 3.3). These values for each usage scenario indicate the optimum performance attainable by the architecture and provide the parameters to guide further exploration by the multiobjective optimizer.

All the previous work on the design space exploration of network processor architectures (such as [4] and [14]) relied on simulation techniques, where different architectures are simulated and evaluated using benchmark workloads. In many cases the search space being explored is very large, and it might be too expensive to evaluate all design alternatives using simulation. The focus of our approach is on a high level of abstraction where the goal is to quickly identify interesting tradeoffs and architectures which can then be further evaluated, by simulation for example, taking finer details into account.

The next section describes our models, following which we describe our method for performance estimation in Section 3. Section 4 briefly introduces the design space exploration scheme and in Section 5 we present our experimental results.

2. MODELING

2.1 Packet Processing

A packet processor operates on interleaved streams of packets which enter the device. All packets belonging to a stream are processed in the same way, i.e. the same set and sequence of tasks are executed for them. In order to know the load on the processing device, it is necessary to have information about the set of tasks associated with a packet stream and the number of packets arriving per time unit. This information can be formalized as follows:

Definition 1. 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 $E(f)$.

In other words, the tasks $V(f)$ must be executed by the packet processor for each packet in stream f. If there is an edge from task t_1 to t_2, then t_1 must be executed before t_2. All tasks can be combined into one conditional task graph where depending on the stream to which a packet belongs, the packet may take different paths through this graph.

As mentioned in the introduction, a packet processing device may be used in a variety of usage scenarios. These scenarios may lead to conflicting design objectives. One of the main ideas in this paper is to exploit of this fact to design architectures which make the best out of possible tradeoffs.

Definition 2. We define a set of scenarios B. To each scenario $b \in B$ there is associated a memory constraint $m(b) \in \mathbb{R}_{\geq 0}$. There is a scenario-stream relation C where $c = (b, f) \in C$ denotes that a stream f is present in scenario b. To each relation c there is associated an end-to-end deadline $d(c) \in \mathbb{R}_{\geq 0}$, a lower arrival curve α_c^l and an upper arrival curve α_c^u.

Here, $m(b)$ denotes the maximum number of packets that can be stored in the processing device in scenario b at any point in time. The end-to-end deadline $d(c)$ with $c = (b, f)$ denotes the maximum allowed time span between the packet entering the processor and the end of the last task execution for a packet of stream f in scenario b. The lower and upper arrival curves are defined as follows.

Definition 3. For any $\Delta \in \mathbb{R}_{\geq 0}$ and any scenario-stream relation $c = (b, f) \in C$, the value $\alpha_c^l(\Delta)$ is smaller than the number of packets arriving in any time interval of length Δ in the stream f in scenario b. In a similar way, the maximum number of packets in an interval of length Δ is always smaller than $\alpha_c^u(\Delta)$.

Therefore, instead of using a stochastic characterization of the input streams, we use deterministic bounds. This approach not only nicely fits into the underlying concept of the SPI model (see [15]) but also follows the usual specification of traffic in the networking context, see the T-SPEC model from IETF [10].

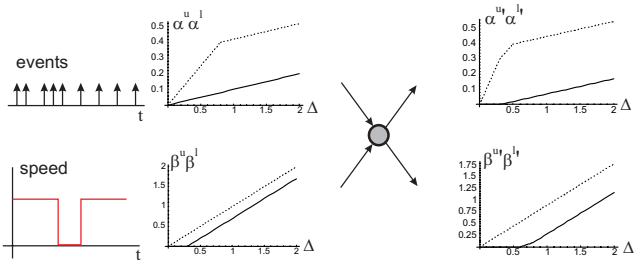

Figure 2: Representation of an arrival curve and a service curve as processed according to Theorem 1.

In Fig. 2, on the upper left corner an example of a packet stream and associated arrival curves is given. Note that in this figure, the arrival curves have been scaled with the computation time for a packet. The steepness of the upper arrival curve until $\Delta = 0.75$ is a measure for the burstiness of packet arrivals.

There are different possibilities to determine the input characterizations of the packet streams:

- The input streams have been characterized in a traffic agreement using T-SPECs.
- Characteristics such as burstiness, burst length, and sustained rates are determined by considering the properties of the generating components, e.g. a sensor and the network connection. In a similar way, properties such as jitter can be easily rephrased in terms of upper and lower arrival curves.
- Measured traffic streams are analyzed with respect to their maximum and minimum number of packets in any time interval.

Methods using the concept of arrival curves to design and analyze communication networks can be found in [5].

In summary, we use a formal means for describing packet rates (using arrival curves), and we gave a model for representing the sequence of tasks to be executed for each packet in a traffic stream, and a model for describing different usage scenarios. In the next section we introduce a representation for packet processing architectures.

2.2 System Architecture

It is widely accepted that packet processing devices will be heterogeneous in nature. In dedicated or application specific instruction set components, simple tasks with high data rate requirements will be executed. Longer and more complicated execution chains will be 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 among packets belonging to different streams [3]. A sketch of the heterogeneous architecture with different packet paths discussed in this paper is shown in Fig. 3 on the left hand side.

Therefore, our model of a feasible system architecture consists of (1) available resource types including their processing capability and performance described by service curves, (2) cost of implementing a resource on the packet processor and (3) the scheduling policies and associated parameters. The logical structure of a system architecture is shown on the right hand side of Fig. 3. Here we see that the processing components have (logically) associated memory which stores packets waiting for the next task to be executed on them. A corresponding scheduling policy selects a packet

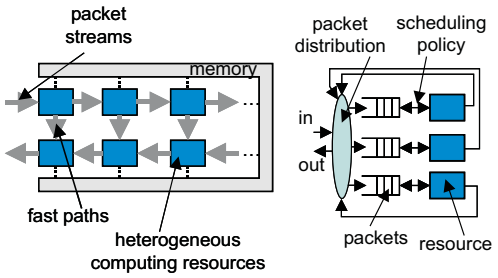

Figure 3: Example of a physical (left) and logical (right) structure of a packet processing architecture.

and starts the execution. The processing of the current packet may be preempted in favor of another task processing a different packet.

Definition 4. 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 type 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. 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 a task t is executed on a resource of type t, then it needs $w(t,s) \in \mathbb{R}_{\geq 0}$ relative computing units for finishing, called the request.

Service curves are defined in a similar way as arrival curves. They describe bounds on the computation capability of the available resource components.

Definition 5. For any $\Delta \in \mathbb{R}_{\geq 0}$ and any resource type $s \in S$, the value $\beta_s^l(\Delta)$ is smaller than the number of available computing units in any time interval of length Δ. In a similar way, the maximum number computing units in any interval of length Δ is always smaller than $\beta_s^u(\Delta)$.

For example, let us suppose that we have a processing resource which is not used by any other task, then it is reasonable to assume that the lower and upper service curves are equal and proportional to Δ: $\beta_s^l(\Delta) = \beta_s^u(\Delta) \propto \Delta$. But if the resource is loaded with some tasks, it is clear that the available computing power within an interval may vary and hence also the time required for executing a task.

An example is given in Fig. 2 on the lower left side. Here, the computing power available is zero in a certain time interval. Therefore, the lower service curve is smaller than the upper one and is zero at the beginning.

Now we can informally describe the design space exploration procedure in the context of packet processors. It selects resources from a given reserve (allocation), associates (or maps) tasks to these resources (binding) and determines a scheduling policy and associated parameters for selecting active packets (scheduling). Although we have chosen a particularly simple cost model, it is not at all clear 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_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 might be intermediate bursts and packet jams.

It is surprising to see, that there exists a *computationally very efficient* possibility to derive *provably correct bounds* on the *end-to-end delays* of packets and on the necessary memory requirements of the processor.

3. ESTIMATING PERFORMANCE AND MEMORY

Our estimation of end-to-end delays and memory requirements is based on the work network calculus [5] from the communication networks area. Recently this approach has been re-formulated in an algebraic setting [2]. In [13], a comparable approach was used to describe the behavior of processing (and not communication) resources. But the resulting formulas for quantities like intermediate arrival and service curves, delays and memory were computationally too expensive to be used in a design space exploration. One of the new results applied in this paper is a simple and efficient piecewise linear approximation. To explain this new methodology, we start with a description of the basic building blocks of a *scheduling network*, describing the flow of packets and resources.

3.1 Basic Building Blocks

The basic idea behind our performance estimation is based on providing a network theoretic view of the system architecture. In particular, packet streams and resource streams flow through a network and thereby respectively change their arrival and service curves. Inputs to a single network node are arrival curves of streams and service curves. Outputs describe the arrival curves of the processed packet streams and the remaining service curves of the (partly) used resources. 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 previous discussion related to Fig. 3, a packet memory is attached to a processing resource which stores the packets that have to wait for being processed. During the processing of a packet there is no remaining computing power, otherwise it is equal to the original one.

In [11], 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 function of the processed event stream:*

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

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

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

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

Note that the arrival function as used above describes bounds on the *computing request* and *not* on the *number of packets*. In Fig. 2, an example of the remaining arrival and service curves is given.

Since we deal with packet streams in the system architecture, we need to convert these streams into corresponding 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

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

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

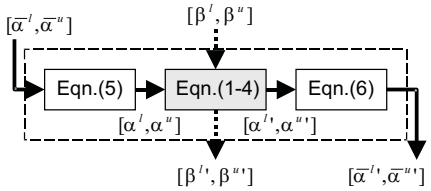

Figure 4: Block diagram depicting the transformation of packet and resource streams by a processing device. The dotted arrows represent the resource flow while the others show the flow of packets and requests.

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

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

The whole transformation is depicted in Fig. 4.

Using this building block, we can describe commonly used scheduling approaches such as *fixed priority preemptive scheduling* [3]. In this case, to each input stream $f \in F$ there is assigned a fixed priority $prio(f) \in \mathbb{Z}_{\geq 0}$. At any point in time, a processing device operates on the packet in its memory (see Fig. 3, 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 modeled directly using our basic building block shown in Fig. 4.

3.2 Scheduling Network Construction

Following the overview of the design space exploration shown in Fig. 1, the scheduling network which enables the estimation of performance measures (such as end-to-end deadlines and memory requirements) is constructed using the specification data in Section 2 and the system architecture provided by the optimization algorithm. In order to simplify the explanation, we restrict ourselves to the use of a fixed priority scheduling policy for all resource types. However, our approach can be generalized to other policies as well, such as GPS [8].

Therefore, the data provided by the multiobjective optimization block in Fig. 1 are as follows.

Definition 6. 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 in a specific usage scenario $b \in B$ is specified by the function $bind(c,t) \in S$ which maps a task in a specific scenario-stream relation $c = (b, f) \in C$ to a resource type s. The scheduling policy is described by a function $prio(c) \in \mathbb{Z}_{\geq 0}$ which associates a priority to each stream in a scenario-stream relation $c = (b, f) \in C$.

Note that a system architecture is not only described by the type and the number of resource components but also by the scenario dependent mapping of tasks to those components.

Now, we can describe the construction of a scheduling network for a given scenario $b \in B$. The basic idea is that the packet streams pass from one resource to the next one. The order is determined by the precedence relations in $E(f)$, see Def. 1. The resource flows, i.e. the capabilities of the resources, also pass through the network. The order is mainly determined by the priorities associated with packet streams.

The procedure for creating a scheduling network is given as follows: Add source and target nodes for all allocated resources. Add source and target nodes for all packet streams f in scenario b. For all tasks $t \in V(f)$, add a scheduling node as shown in Fig. 4 to the scheduling network and connect its packet stream input/output according to the precedence relations in $E(f)$. Connect the resource stream input/output of the scheduling nodes according to the precedence relations and according to the stream priorities in scenario b.

As a result of applying this procedure we get a scheduling network for each scenario, containing source and target nodes for the different packet streams, and resource flows and scheduling nodes which represent the computations described in Fig. 4. A concrete example to illustrate this is given in Fig. 9.

Given the arrival curves for all source packet nodes, i.e. $[\alpha_c^l, \alpha_c^u]$ with $c = (b, f)$ for stream f in scenario b, and the initial service curves for the allocated resources, i.e. $[alloc(s)\beta_s^l, alloc(s)\beta_s^u]$ for resource type s with $alloc(s)$ 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 Performance Optimization

In order to estimate the properties of a packet processing system architecture we need quantities like bounds on end-to-end delays of packets to be processed and on 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 packets waiting to be processed.

The following two equations give bounds on both quantities:

$$delay_c \leq \sup_{u \geq 0} \left\{ \inf\{\tau \geq 0 : \alpha_c^u(u) \leq \beta_c^l(u+\tau)\} \right\} \tag{7}$$

$$backlog_c \leq \sup_{u \geq 0} \{\alpha_c^u(u) - \beta_c^l(u)\} \tag{8}$$

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 have to know which curves to use in (7) and (8). For α_c^u, we need the initial upper arrival curve of an incoming packet stream f. The service curve β_c^l to be used is the *accumulated* curve of all scheduling nodes through which the packets of stream f pass in the scenario b. As described in [2], this quantity can be determined through an iterated convolution.

Finally, we need to determine some measure for the performance of a given architecture under a certain usage scenario. Therefore, we introduce a parameter ψ_b, which is the largest scaling factor of the packet input streams f with $c = (b, f) \in C$ according to $[\psi_b \alpha_c^l, \psi_b \alpha_c^u]$, such that the constraints on end-to-end delays and memory are still satisfied.

3.4 Piecewise Linear Approximation

The only problem that we are left with is the efficient computation of the expensive equations in Theorem 1. Note that this set of equations has to be computed for all scheduling nodes in any scheduling network. In addition, if the design space exploration is to be based on evolutionary multiobjective algorithms, the performance of many system architectures need to be estimated.

To this end, we propose a piecewise linear approximation of all arrival and service curves. The approximation consists of a combination of two line segments and is shown in Fig. 5. As a shorthand notation we can denote curves γ^u and γ^l by the tuples $U(q,r,s)$ and $L(q,r,s)$, respectively.

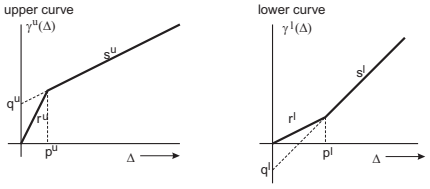

Figure 5: Simple representation of upper and lower curves.

Based on this approach, the equations in Theorem 1 and the equations (7) and (8) can be computed *symbolically*. We omit the explicit formulas because of space restrictions.

Referring to our initial Figure 1, we have now finished the description of the input specification, the construction of the scheduling network, and the performance optimization. It remains to be shown, how a multiobjective optimization algorithm can be used to perform the exploration task and come up with candidate architectures which are then evaluated using our scheme.

4. MULTIOBJECTIVE DESIGN SPACE EXPLORATION

As described in the introduction, in the case of network processor design we are faced with conflicting objectives. Here we show how to obtain a tradeoff between the performance ψ_b in different usage scenarios b and the cost of the system architecture

$$cost = \sum_{s \in S} alloc(s) cost(s) \tag{9}$$

We can formulate this task as a multiobjective optimization problem where with each implementation there is an associated objective vector $v = (v_0, ..., v_{k-1})$ with k elements. The goal is to determine implementations with Pareto-optimal objective vectors. The following definition supposes that we are trying to find vectors with small elements, i.e. we are considering a minimization problem. Note that our design space exploration problem can be formulated in this way by, for example, using $v_0 = cost$ and $v_i = 1/\psi_{b_i}$ for all $b_i \in B$, $i > 0$.

Definition 7. 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 k, we have $v_k < g_k$. A vector is called Pareto-optimal if it is not dominated by any other vector in V.

The architectures with Pareto-optimal objective vectors represent the tradeoffs in the network processor design. There are many different approaches to multiobjective optimization, e.g. Tabu search, simulated annealing or evolutionary algorithms [6]. We have chosen evolutionary algorithms, but other options also could have been used.

The evolutionary multiobjective optimizer that we have used, called SPEA2 [6], maintains a population of current solutions and an archive which stores the best solutions found so far. Independent of the optimization principle used, the optimizer generates new solutions (system architectures) based on the already known set. These new solutions are then evaluated for their objective vector. The following methods are used in order to include domain specific knowledge into the search process:

(1) System architectures are represented according to Def. 6. In particular, the representation contains the allocation $alloc(s)$ for all resource types $s \in S$, the binding of tasks to resources *for each scenario*, and the stream priorities *for each scenario*.

(2) New system architectures are determined using *mutation* and *crossover* operations [6]. In case of the mutation, either allocation, or the binding for a scenario, or the priorities are mutated. In case an infeasible system architecture is generated (see Section 3.2) a repair strategy is invoked which attempts to maintain a high diversity in the population. In a similar way, crossover combines two selected solutions to generate a new one by combining either the allocations, the bindings of a selected scenario, or the priorities of a selected scenario.

Clearly, because of this heuristic search procedure, no statements about the optimality of the final set of solutions can be made. There is experimental evidence, that the solutions found are close to optimal even for realistic problem complexities.

5. EXPERIMENTAL RESULTS

Our input traffic consists of a mixture of real-time (RT) and non-real-time (NRT) streams. We have considered two scenarios b_1, b_2 where the first scenario b_1 models a configuration in a network backbone where aggregates of streams are forwarded with the smallest possible computation requirement but at high bandwidths. This scenario therefore contains a single stream class for packet forwarding (NRT_Forward). The second scenario b_2 however resembles an access link from a customer to a service provider. Typically an access network has relatively low bandwidth but high computation requirements. The computations are in particular required to perform encryption, e.g. Triple-DES [1] to allow sensitive information to be transmitted over untrusted networks of a service provider. Another application is voice coding/decoding (e.g. in accordance with ITU G.723.1) to use telephony services over the Internet. Of course, a network processor for an access link could also be used to simply forward traffic from the customer to the service provider. Hence, we define a set of five traffic streams $F = \{$NRT_Forward, RT_Send, RT_Recv, NRT_Encrypt, NRT_Decrypt$\}$, containing two real-time flows based on voice services and three flows with relaxed quality of service requirements where two of them apply Triple-DES. To process these five streams we define 25 tasks, i.e. $|T| = 25$. The task graph and the dependencies are shown in Fig. 6.

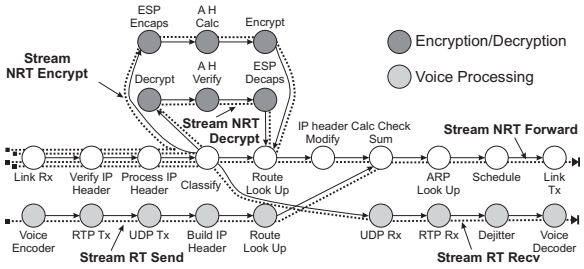

Figure 6: Task graph for a network processor.

The scenario-stream relation C can be represented as in Fig. 7.a). Here, one can also see an example of an end-to-end deadline $d(c)$, a memory constraint $m(b)$, and arrival curves $[\alpha_c^l, \alpha_c^u]$ represented in the piecewise linear form defined in Section 3.4.

Further, we use 8 different resource types s with $S = \{$Classifier, PowerPC, ARM9, MEngine, CheckSum, Cipher, DSP, LookUp$\}$. Each one of these has different computation capabilities that can be represented in form of the mapping relation M, see Def. 4. Part of this specification is represented in Fig. 7.b), including an example for the implementation cost $cost(s)$, number of instances $inst(s)$ and request $w(t,s)$. The initial service curves are simply set to $[\beta_c^l, \beta_c^u] = [L(0,0,1), U(0,0,1)]$, i.e. $\beta_c^l(\Delta) = \beta_c^u(\Delta) = \Delta$, reflecting the fact that the resources are fully available for the processing of the tasks.

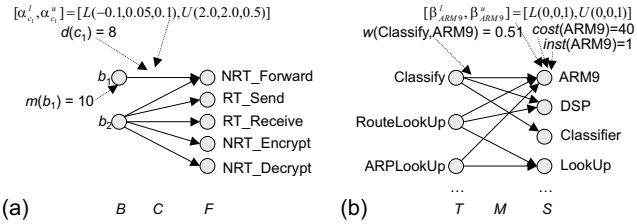

Figure 7: Graphical representation of (a) the scenario-stream relation C and (b) a part of the task-resource relation M.

Fig. 8 (a screen shot from our tool) shows the initial *population* on the left and the final population of a specific design space exploration run on the right. Each dot represents one system architecture (resources, binding and scheduling policy). The axes represent the total cost of the implementations (see Eqn. (9)) and the maximum scaling factor for scenario b_1 of the packet input streams until either

the end-to-end deadline or the maximum memory is reached. Note that we are looking at projections of the three-dimensional objective space on two dimensions. Hence, we do not see the "typical" Pareto-tradeoff curves. Nevertheless, no objective vector (or system architecture) dominates another one (see Def. 7). We will not discuss the details of the tradeoff curves, but would like to point out that the different branches that can be seen in the projection are caused by the availability of specialized hardware components which are only useful for some of the input streams.

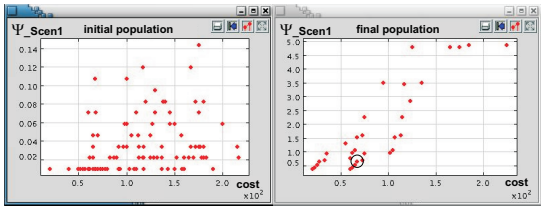

Figure 8: Initial population and final population of a design space exploration run.

Finally, let us look more closely at one of the implementations. It is marked in Fig. 8 and its objective vector consists of $cost = 67$, the maximal scaling for the input streams for scenario 1 (b_1) is $\psi_{b_1} = 0.644$ and that for scenario 2 (b_2) is $\psi_{b_2} = 0.400$. The allocated resource components are CheckSum, Cipher, ARM9 and LookUp. Therefore, we have $alloc(\text{CheckSum}) = alloc(\text{Cipher}) = alloc(\text{ARM9}) = alloc(\text{LookUp}) = 1$ (see Def. 6). In scenario 2 (b_2) all the 5 streams in F are present. The ordering of the priorities as determined by the design space exploration for this scenario is: NRT_Encrypt, NRT_Forward, RT_Send, RT_Receive and NRT_Decrypt. Here the stream NRT_Encrypt has priority 1 (i.e. $prio(\text{NRT\_Encrypt}) = 1$) and NRT_Decrypt has priority 5. Due to space restrictions, not all bindings are given here. The task VerifyIPHeader is mapped to the CheckSum resource, tasks ProcessIPHeader, Classify, VoiceEncoder, RTPTx, UDPTx, BuildIPHeader are executed on the ARM9 resource and tasks RouteLookUp are mapped to the LookUp resource. Based on this information, the system internally builds the scheduling graph and evaluates it according to the methods described in Section 3. Part of the scheduling network (see Section 3.2) is represented in Fig. 9.

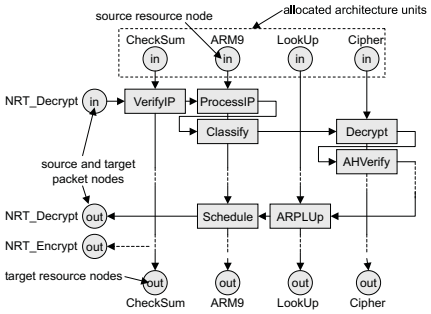

Figure 9: Part of a schedule network constructed for the performance estimation of a given system architecture. The scheduling policy is *fixed priority* and the internal scheduling nodes correspond to the basic blocks shown in Fig. 4.

Our tool performed the above described design space exploration. It uses the evolutionary optimization tool SPEA [6], and an implementation of performance optimization, population and archive handling, mutation, crossover and repair in Java. The graphical output used the Ptolemy-Plot routine from UC Berkeley [9], the graphical input and tool integration was based on the modeling and simulation tool MOSES [7]. The experiments have been run on a Pentium III under LINUX. The population size was 100 system architectures and the optimization was stopped after 300 generations. Each generation, i.e. the mutation, crossover and performance optimization (including scheduling network construction) takes about 2 sec which leads to an overall optimization time of 10 minutes.

6. CONCLUDING REMARKS

We presented several new results concerning the modeling and design space exploration of packet processing devices, such as network processors. Our work involves a careful combination of some known and some new solutions: (1) modeling the flow of packet streams through a heterogeneous system architecture consisting of computation and communication components using a scheduling network, (2) using a real-time calculus to determine end-to-end deadlines and memory requirements and using piecewise linear approximations for fast performance estimation, and finally (3) combining these models and methods in a design exploration system for packet processors.

These results are based on several abstractions. In particular, we neglect effects of caching, separate memories, and shared communication resources in the processor architecture. The extension of the presented work to incorporate these additional constraints is a subject of further investigation.

Acknowledgement

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

7. REFERENCES

[1] American National Standards Institute. *Triple Data Encryption Algorithm Modes of Operation, ANSI X9.52-1998*, 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. Information Theory*, 37(1):114–141, 1991.

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

[7] MOSES project. `www.tik.ee.ethz.ch/~moses/`.

[8] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks. *IEEE/ACM Trans. Networking*, 2-2:137–150, 1994.

[9] Ptolemy project. `ptolemy.eecs.berkeley.edu/`.

[10] S. Shenker and J. Wroclawski. General characterization parameters for integrated service network elements. RFC 2215, Internet Engineering Task Force (IETF), Sept. 1997.

[11] 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*, LNCS 2211. Springer-Verlag, 2001.

[12] 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*, volume 4, pages 101–104, 2000. Invited paper.

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

[14] 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.

[15] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI – a system model for heterogeneously specified embedded systems. *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, 2002. Accepted for publication.