# Pipelining vs. Multiprocessors – Choosing the Right Network Processor System Topology

Ning Weng and Tilman Wolf
*Department of Electrical and Computer Engineering*
*University of Massachusetts at Amherst*
*{nweng,wolf}@ecs.umass.edu*

## Abstract

*Computer networks provide an increasing number of services that require complex processing of packets, for example firewalls, web server load balancing, network storage, and TCP/IP offloading. Such services are typically limited to the network edge, but still require the use of powerful network processors to provide the necessary performance and flexibility. To keep up with these demands, next generation network processors will provide dozens of embedded processor cores. One important question is which system-on-a-chip topology is most suitable for such a highly parallel system. Due to the inherent parallelism in network traffic, the design space for this problem is vast and ranges from pipelined to multiprocessor solutions. In this work, we present a methodology to explore this design space through performance modelling. It is crucial that such a model considers the NP workload in detail as the distribution of processing steps over the system has significant impact on system throughput. We present a mechanism for mapping the workload optimally to an arbitrary topology based on run-time traces. Our performance model determines the system performance and considers the effect of on-chip communication as well as off-chip memory accesses. We present results from several NP applications, show the performance tradeoffs between different system topology, and identify system bottlenecks.*

## 1 Introduction

The Internet has progressed from a simple store-and-forward network to a more complex communication infrastructure. In order to meet demands on security, flexibility and performance, network traffic not only needs to be forwarded, but also processed inside the network. Such processing is performed on routers, where port processors can be programmed to implement a range of functions from simple packet classification (e.g., for firewalls) to complex payload modifications (e.g., encryption, content adaptation for wireless clients, or ad insertion in web page requests).

To provide the necessary flexibility for increasingly complex network services at the edge, router designs have moved away from the hard-wired ASIC forwarding engines. Instead, programmable "network processors" (NPs) have been developed in recent years. These NPs are typically single-chip multiprocessors with high-performance I/O components. Network processor are usually located on a physical port of a router. Packet processing tasks are performed on the network processor before the packets are passed on through the router switching fabric and through the next network link. This is illustrated in Figure 1.

To meet the performance demands of such a system it is important to exploit the inherent parallelism of network processing. Due to limitations in clock rates and memory access speeds, a single processor cannot meet the processing requirements for a Gigabit router system. Instead, network processors use multiple processing engines that process packets in a parallel of pipelined fashion. Most packets in network traffic do not exhibit interdependencies and thus can be processed independently. This allows network processors to employ highly parallel architectures that are optimized for packet throughput. This is very different from the goals of workstation or server processors, which aim at optimizing single-task performance.

One of the key architectural aspects of a network processor is the system topology that determines how processing engines are interconnected and how parallelism is exploited. This is an important characteristic as it determines the system performance as well as the overall scalability of the architecture. In principle, there are several approaches to arranging processing engines: in parallel, in a pipeline, or a hybrid configuration. The goal of this paper is to model and quantify the system performance of these approaches. The results allow us to understand the tradeoffs between different topology, identify optimal solutions, and find bottleneck components in the system. The impact of this exploration is illustrated in Figure 2, which shows the throughput speedup of a network processor system versus the number of processors in the system. The details on how the results are derived are explained later in the paper. A given number of processors is arranged in any possible configuration ranging from parallel processors to a complete pipeline. The throughput of the configuration is then evaluated. The main observation is that the performance for a given number of processors can vary by a factor of 2-3. This indicates that the choice of topology does have a considerable impact on the overall performance. Another im-

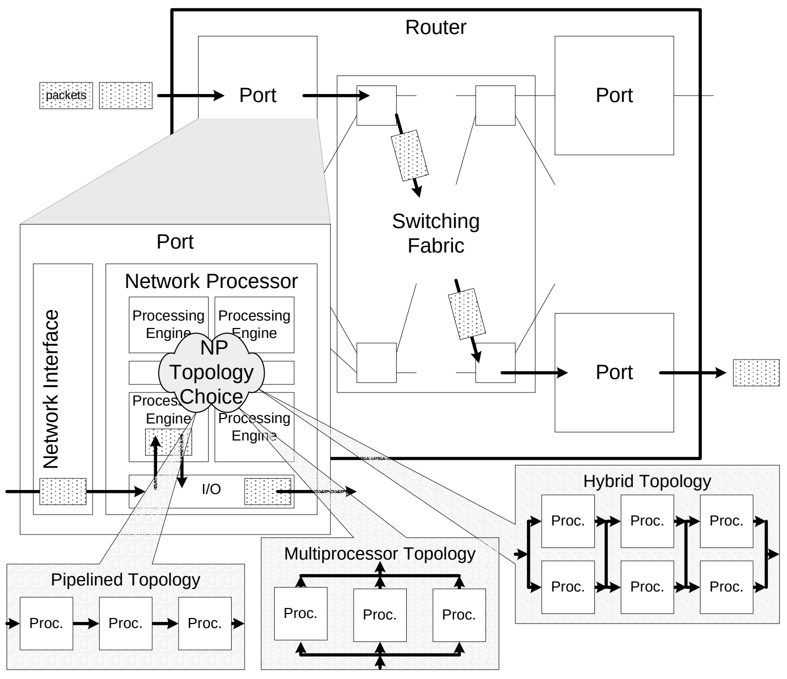


Figure 1: Router System with Network Processors. The processing elements on the network processor can be arranged in various topologies.

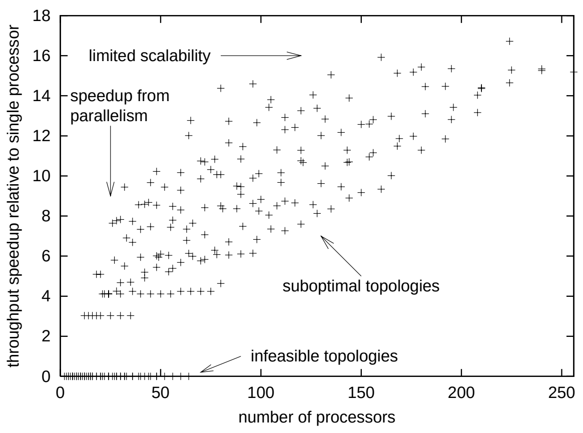


Figure 2: Topology Impact Performance. The system throughput depends heavily on the choice of topology. The results are obtained from the performance model presented in Section 5.

portant observation from Figure 2 is that the scalability of these topologies is limited. For 100 processors, the maximum throughput is only slightly higher than for 50 processors. The performance model that we derive in our work can help identify system bottlenecks (e.g., memory interfaces) and quantify their performance impact.

In order to explore different NP architectures and quantitatively obtain the performance of a particular architecture, we need to consider the impact of the network processor workload. The workload (consisting of a set of "applications") determines the processing steps that need to be performed on each packet and thus the overall performance of the system. Our approach to modeling network processor throughput under different topologies and workloads consists of the following three steps:

1. **Abstract Application Representation.** The workload needs to be represented in a way that it can be mapped to multiple parallel or pipelined processing elements. This requires a representation that exhibits the application parallelism while also ensuring that data and control dependencies are considered. We use an annotated directed acyclic graph (ADAG) to represent the dynamic execution profile of applications.

2. **Application Mapping.** Once we have the application represented as an ADAG, the next step is to map the ADAG onto a NP topology. The goal is to optimize system performance while considering all the application dependencies. To approximate a solution to this NP complete problem, a randomization algorithm is used that achieves good results.

3. **NP System Performance Modeling.** The system performance model is the key component for the design space exploration of different NP topologies. This model considers processing, inter-processor communication, contention on memory interfaces, and pipeline synchronization effects to determine the overall throughput.

We obtain results from an exhaustive evaluation of the entire design space. The results that we obtain can be used in a number of ways:

- Understanding the trade-offs involved in selecting different network processor topologies given a particular workload.

- Understanding how to optimize and map applications given a particular NP architecture.

- Modeling throughput performance of network processor architectures.
- Identify the bottlenecks of the system and determining their quantitative impact on system performance.

The remainder of this paper is organized as follows. We discuss work related to design space exploration and performance modelling of NP architectures in Section 2. Section 3 describes our methodology and generalized network processor topology for exploring the design space. Section 4 introduces the ADAG abstraction to describe the workload and map it to the NP topology. The analytic performance model for evaluating an NP system is presented in 5. Results are presented and discussed in Section 6. Section 7 summarizes and concludes the paper.

## 2 Related Work

The system topology of network processors can be implemented in a variety of ways, ranging from a fully pipelined systems to fully pooled multiprocessors. An examples for the pipelined system is the Agere Fast Pattern Processor and Routing Switch Processor [19] and for the fully pooled system the Intel IXP2400 [14]. Between them lie the hybrid approaches, or pipeline of pools, with EZchip's NP-1 [7] as an example. This topic has been explored by Gries *et al.* [11], who evaluated the performance/cost trade-offs for different network processor topologies based on a network calculus approach. The major difference to our work is that we develop a mean value analysis using detailed instruction traces from network processor workload simulations.

The design spaces of network processors has been explored in several ways. Crowley *et al.* have evaluated different processor architectures for their performance under different networking workloads [5]. This work mostly focuses on the tradeoffs between RISC, superscalar, and multithreaded architectures. In more recent work, a modeling framework is proposed that considers the data flow through the system [4]. Thiele *et al.* have proposed a performance model for network processors [29] that considers the effects of the queueing system. In our previous work, we have developed a quantitative performance and power consumption model for a range of design parameters [8] [9]. All these models require detailed workload parameters to obtain realistic results.

Several network processor benchmarks have captured the characteristics of network processing. Crowley *et al.* have defined simple programmable network interface workloads [5]. In our previous work, we have defined a network processor benchmark called CommBench [32]. Memik *et al.* have proposed a similar benchmark more recently [21]. All these benchmarks are useful in that they define a realistic set of applications, but are limited in how much detail of workload characteristics can be derived. We have addressed this shortcoming with a very detailed and network-processor-specific analysis of such benchmark applications that goes beyond simple microarchitectural metrics. In our recent work, we have developed PacketBench [26], which is a network processing simulation environment that can derive detailed instruction traces of packet processing applications. These run-time traces can be used to explore simple application metrics as well as more complex processing task dependencies. These dependencies are important information when distributing processing task to multiple processing engines.

Mapping of task graphs to multiprocessors is a well explored research area, which has been surveyed by Kwok and Ahmad [17]. Since finding an optimal mapping is an NP-complete problem in general, researchers have relied on devising efficient heuristics. However, it is in a general conflicting goal to design a mapping algorithm with low computation time but high quality. One promising approach is to use randomization. As indicated by Karp [15] and Motwani and Raghavan [22], an optimization algorithm which makes random choices can be very fast and simple to implement. Recently, Lakamraju *et al.* [18] have applied this idea to synthesizing networks that satisfy multiple requirements. Their result demonstrates that randomization is a powerful approach that can generate surprisingly good results. Also, Kwok [16] has used random neighborhood searching techniques to propose a Bounded Number of Processors (BNP) scheduling algorithm. We also use a randomized algorithm for our task mapping problem.

The delay effect caused by the memory interference in a parallel processing system with a shared memory has been well investigated. Specific solutions are provided in [2] [12] [10]. Methods have also been developed that provide lower bounds on response times for parallel systems with deterministic service time [30].

## 3 Network Processor Design Space Exploration

The modeling of network processor performance requires an abstract representation of NP architectures. We introduce a general NP architecture that allows us to analytically model the NP performance and compare different architectures.

### 3.1 General Network Processor Topology

Network processors exhibit a large diversity in architecture with system topologies ranging from simple multiprocessors to complex data flow pipelines. In order to explore this large design space in a general way, we introduce a network processor architecture that features the main characteristics of NPs while abstracting away some of the system details.

We use a general, parameterized network processor topology shown in Figure 3. The system topology is characterized by three components: processing elements,

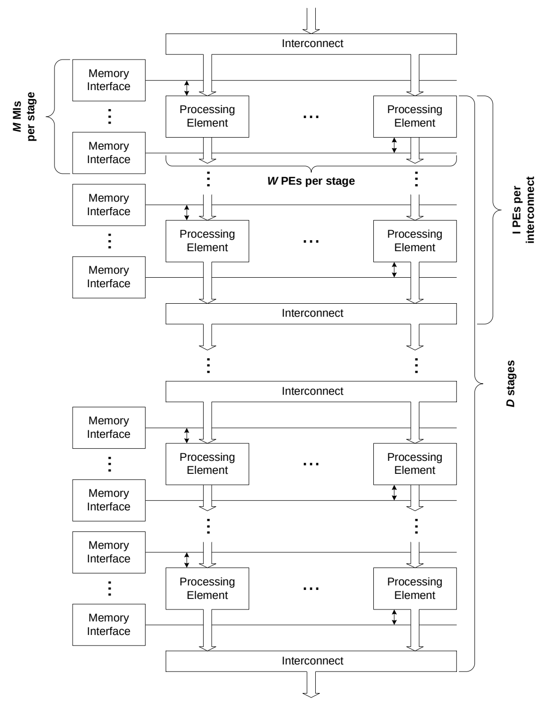


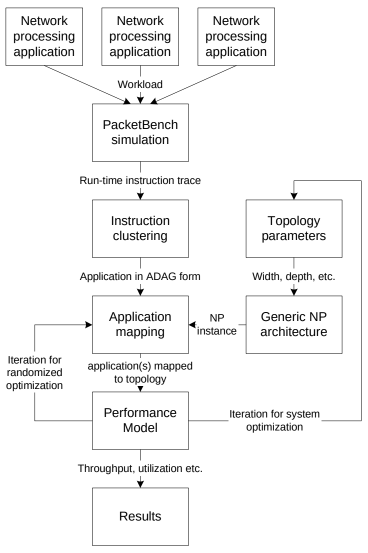
Figure 3: Generic Network Processor Architecture. The parameters which can be varied are shown in the figure.

shared interconnects, and memory interfaces. The packets move from top to bottom. The key parameters are:

- The width of the pipeline stage ($W$).
- The depth of the pipeline stage ($D$).
- The number of stages per communication interconnect ($I$).
- The number of memory channels shared by one row of processing elements ($M$).

The flexibility of these parameters enables our generic NP architecture to represent a wide range of possible NP architectures. A parallel multiprocessor based topology can be modelled by fixing the pipeline depth ($D$) and interconnect stages ($I$) to 1, while varying the pipeline width ($W$). A pipeline architecture can be modelled by setting both pipeline width ($W$) and pipeline interconnects ($I$) to 1, while varying the pipeline depth ($D$). The number of available memory channels per stage can range from 1 to $W$.

This model can represent a large range of NP topologies, but it needs to be noted that not all possible combinations are feasible. The available chip area limits the number of processors and interconnects; available power limits the processor clock rates; and packaging technologies limits the number of available pins for off-chip memory interfaces. We have derived analytic models to account for these costs in prior work [8] [9]. In this work, we do not directly address these technology limitations as the main objective is to develop a fundamental understanding of NP topology tradeoffs.



Network processing application
Network processing application
Network processing application
Workload
PacketBench simulation
Run-time instruction trace
Instruction clustering
Topology parameters
Application in ADAG form
Width, depth, etc.
Application mapping
NP instance
Generic NP architecture
Iteration for randomized optimization
application(s) mapped to topology
Performance Model
Iteration for system optimization
Throughput, utilization etc.
Results

Figure 4: Design Space Exploration Methodology.

## 3.2 Methodology

The methodology used to explore the design space is shown in Figure 4. The central components of the methodology are the application mapping onto the NP architecture and the performance model.

The input of the application mapping is an abstract representation of the application. We use a directed acyclic graph to represent this application processing. This ADAG is derived from the PacketBench simulator, which simulates NP applications and generates run-time instruction traces. The mapping algorithm then assigns processing resources to each processing step of the application. For each mapping result, the performance of the overall system can be determined through the performance model. Since the mapping problem is NP-complete, a randomized mapping algorithm is used to iterate over a number of possible mappings and to derive a close to optimal system performance. By varying the topology and system parameters, a large number of NP system topologies can be evaluated.

The results that can be derived for a given application or workload, are the optimal topology parameters (width of pipeline, depth of pipeline, and number of memory channels). In addition to point solutions, parameter tradeoffs can be explored (e.g., throughput of system over all combinations of width and depth).

The following sections explain the application representation, mapping, and performance model in more detail.

## 4 Workload Representation and Mapping

A parallel and pipelined network processor provides numerous processing resources to which processing tasks can be assigned. In order to model and evaluate NP topologies, it is necessary to derive an abstract description of the workload such that it can be mapped to any given NP. This representation should capture the run-time properties of the application at sufficient detail to allow the identification of parallelizable application parts. We have chosen an annotated directed acyclic graphs (ADAG) for this purpose.

## 4.1 ADAG Generation

Our approach to generating the ADAG is to analyze the run-time characteristics of NP applications and build the application representation "bottom-up." The annotations indicate the processing and I/O requirements of each block and the strength of the dependency between blocks. We consider each individual data and control dependency between instructions and group them into larger clusters, which make up the ADAG.

### 4.1.1 Run-Time Trace Generation

In order to obtain runtime analysis of application processing, we use a tool called "PacketBench" that we have developed [26]. The goal of PacketBench is to emulate the functionality of a network processor and provide an easy to use environment for implementing packet processing functionality. One key benefit of PacketBench is the ease of implementing new applications. The architecture is modular and the interface between the application and the framework is well defined. New applications can be developed in C, plugged into the framework, and run on the simulator to obtain processing characteristics.

The PacketBench executable is simulated on a typical processor simulator to get statistics of the number of instructions executed and the number of memory accesses made. We use the ARM target of the SimpleScalar [3] simulator, to analyze our applications. This simulator was chosen because the ARM architecture is very similar to the architecture of the core processor and the microengines found in the Intel IXP1200 network processor [13], which is used commonly in academia and industry. The tools were setup to work on an Intel x86 workstation running RedHat Linux 7.3. PacketBench supports packet traces in the *tcpdump* [28] format and the Time Sequenced Header (TSH) format from NLANR [23].

The run-time traces that we obtain from PacketBench contain the instructions that are executed, the registers and memory locations that are accessed, and an indication of any potential control transfer. Using these traces we build an ADAG that considers dependencies among instructions as well as allows us to discover any potential parallelism. The trace also provides detailed memory access statistics which are essential to obtain accurate results using our performance models. Since we make no assumption on the processing order other than the dependencies between data, we are able to represent the application almost independently from a particular system.

This approach differs from conventional, static call-graph analysis. One of the key points of this work is that the ADAG can be created *completely automatically* from a run-time instruction trace of the program on a uniprocessor system. Once this is done, the iteration over the design space can be performed rapidly unlike trace based evaluation which would require a new trace to be generated for each variation in architecture. This is due to the fact that our ADAG is an architecture independent representation of the processing required by an application. The drawback to using a dynamic instruction trace is that each packet could potentially execute a different set of instructions depending on the values contained within. Since most packets do exhibit the same processing trends (i.e., fast path processing), we believe that the inaccuracy introduced by using a dynamic trace does not impact the overall results.

### 4.1.2 ADAG Clustering

The ADAGs that are directly derived from the instruction traces contain a large number of instructions and basic blocks (i.e., nodes in the ADAG). Their processing requirements can range from a single instruction to several dozen. In order to make the ADAGs more tractable and allow an efficient mapping process, we use an ADAG clustering algorithm to group instruction sequences. The resulting ADAG contains a smaller number of instructions, but still reflects all the dependencies, processing, and I/O requirements of the original ADAG.

Finding an optimal clustering of the ADAG nodes is an NP-complete problem. We have developed a heuristic called "Maximum Local Ratio Cut" (MLRC) [25], which is based on the "ratio cut" metric introduced by Wei and Cheng [31]. This clustering method achieves a natural clustering of nodes through minimizing the inter-cluster dependencies, while maximizing cluster size of cohesive nodes.

For the purpose of exploring pipelined network processor topologies, it is crucial that the processing tasks in each pipeline stage are roughly balanced. The synchrony enforced by the pipelining paradigm reduces the pipeline speed to the slowest stage. Large differences in the amount of processing will cause an inefficient NP system. In order to enforce such a balance, we further constraint the MLRC algorithm to minimize the proportions between the largest and the smallest processing task.

### 4.1.3 Sample Applications

For this work, we explored four applications that are typical of network processor workloads. These applications are:

- **IPv4-radix.** IPv4-radix is an application that performs RFC1812-compliant packet forwarding [1] and uses a radix tree structure to store entries of the routing table.

- **IPv4-trie.** IPv4-trie is similar to IPv4-radix and also performs RFC1812-based packet forwarding. This implementation uses a trie structure with combined level and path compression for the routing table lookup [24].

- **Flow Classification.** Flow classification is a common part of various applications such as firewalling, NAT, and network monitoring. The packets passing through the network processor are classified into flows which are defined by a 5-tuple consisting of the IP source and destination addresses, source and destination port numbers, and transport protocol identifier.

- **IPSec Encryption.** IPSec is an implementation of the IP Security Protocol, where the packet payload is encrypted using the Rijndael algorithm [6], which is the new Advanced Encryption Standard (AES).

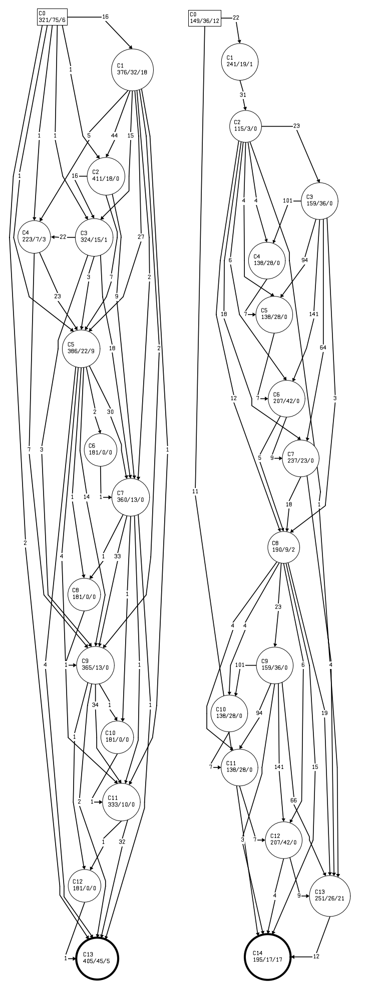
After simulating these applications, collecting runtime traces, identifying data and control dependencies, and clustering instructions to processing tasks, we obtain the ADAGs shown in Figure 5. A node represents a processing task and an edge shows a dependency (control and/or data). The annotation in a node consist of the node name and a 3-tuple that contains the number of instructions that are executed, the number of reads to memory and the number of writes to memory. The reads and writes to memory are only for data that is used for the first or the last time by the application. Data that is moved between processing tasks is shown as edge weights on the dependencies. Rectangular nodes identify the starting node, bold nodes indicate the terminating nodes.

The ADAGs display the inherent characteristics of the applications. IPv4-radix and IPSec highly sequential applications, which is due to data dependent processing. IPv4-trie and Flow Classification show a bit more parallelism.

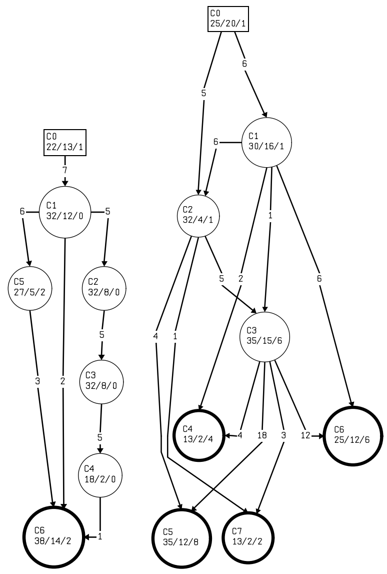
## 4.2 ADAG Mapping

The mapping process is the first step in the actual design space exploration. The goal is to map one or multiple ADAGs to an instance of the generic NP system topology in such a way as to maximize the optimization criterion. This is not an easy task because the mapping process needs to consider the dependencies within an ADAG and ensure that a correct processing of packets is possible. Further, the mapping can have significant impact on the overall system performance. In a pipelined system, the performance depends on the speed of the slowest pipeline stage. By distributing processing tasks evenly over the available processing elements, a better system performance can be achieved. In this work, we use the system throughput as the performance metric. The throughput is determined by the performance model described Section 5 and depends on the processing that is performed in each stage of the pipeline, the communication between stages, and the delay caused by off-chip memory accesses.

Unfortunately, the problem of finding an optimal mapping solution is NP complete. Malloy *et al.* established that producing a schedule for a system that includes both execution and communication cost is NP-complete, even if there are only two processing elements [20]. Therefore



(a) IPv4-radix (b) IPSec



(c) IPv4-trie (d) Flow Classification

Figure 5: Annotated Directed Acyclic Graphs (ADAGs) for Workload Applications. The annotation in a node is the node name and a 3-tuple (processing/reads/writes). The weight on the edges is the number of data and control dependencies between nodes.

we need to develop a heuristic to find an approximate solution.

Our heuristic solution to the mapping problem is based on *randomized mapping*. The key idea is to randomly choose a valid mapping and evaluate its performance. By repeating this process a large number of times and picking the best solution that has been found over all iterations, it is possible to achieve a good approximation to the global optimum. The intuition behind this is that any algorithm that does not consider all possible solutions with a non-zero probability might get stuck in a local optimum. With the randomized approach any possible solution is considered and chosen with a small, but non-zero probability. This technique has been proposed and successfully used in different application domains by Karp [15], Motwani [22], and Lakamraju *et al.* [18].

We break the mapping algorithm into two stages: mapping and filtering. In the mapping stage, we randomly allocate an ADAG node (i.e., a processing task) to an empty processing element in the NP topology. If the mapping violates the dependency constraints, we repeat the mapping until a valid mapping has been found or a certain number of attempts has been reached. This is repeated until all nodes of an ADAG have been placed into the topology. The ADAG mapping is repeated until the NP topology is "full" (i.e., no ADAG can be added successfully). The mapping is then moved to the filtering stage where the performance of the mapping is determined and compared to prior maps. If the new mapping is the best solution in terms of the optimization metric it is recorded for comparison to future solutions. Otherwise it is discarded. At the end of the mapping process, the best overall mapping is reported.

The execution time and accuracy of the mapping algorithm is determined by the parameters that determine how persistent the algorithm is about finding a solution. Our solution uses two such parameters: the maximum *node failure counter* and the maximum *ADAG failure counter*. The *node failure counter* counts how many times it has been attempted to map a particular node to the NP topology. Due to topology and ADAG dependency constraints, a valid mapping might not be possible. Thus, when the maximum value for the *node failure counter* has been reached, the attempt to map the ADAG to the topology is aborted. This event counts as an *ADAG failure*. When the maximum number of *ADAG failures* have been reached, the the topology is considered full and the mapping is passed on to the filtering stage. The *node failure counter* and the *ADAG failure counter* are reset after each successful mapping of a node or an ADAG respectively.

To reduce the number of unsuccessful mapping attempts, certain additional heuristics are employed. For example, the child nodes of a given ADAG node can only be mapped to processing stages after the current node (otherwise there is an immediate dependency violation). Thus, the random choice of processing elements is limited to those stages. This does not impact the correctness of the algorithm, but improves the probability that a better mapping is found.

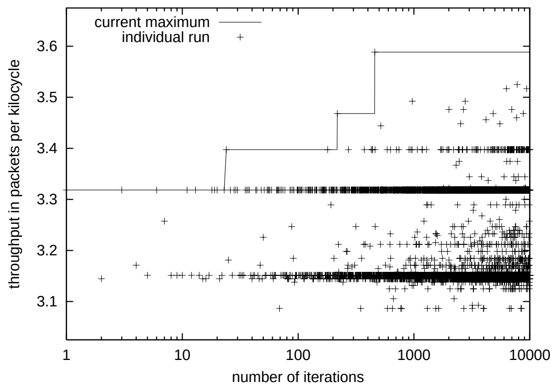


Figure 6: Randomized Mapping. Throughput over 10,000 randomized mapping runs.

Figure 6 shows the range of solutions that are derived in this process. The x-axis shows the iteration of the mapping algorithm. The y-axis show the performance of a particular solution. The solid line indicates the best solution up to a given iteration that has been found in the filtering stage. The logarithmic scale of the x-axis highlights that a large number of iterations are necessary to get incrementally better results once a few hundred or thousand iterations has been exceeded.

Altogether, randomized mapping is a very suitable approach to solve the problem of NP completeness of mapping ADAGs to complex NP topologies. Since we use analytic performance modeling rather than simulation to determine the throughput of a system, we can quickly determine the quality of a mapping solution and iterate over a large number of possible solutions. This would not be possible if more time consuming simulation based modeling was used.

## 5 Performance Model

The performance model for the general NP topology provides an analytic expression for the system throughput. After mapping the application ADAGs to the network processor topology, we know exactly the workload for each processing element. This information includes the total number of instructions executed, the number of memory accesses, and the amount of communication between stages. Thus, the model needs to take this into account as well as contention for shared resources (memory channels and communication interconnects).

### 5.1 Throughput

The system throughput is determined by modeling the processing time for each processing element in the system based on its workload, the memory contention on each memory interface, and the communication overhead between stages. We are particularly interested in the maximal latency, $\tau_{stage_{max}}$, since the critical stage of the pipeline architecture determines the overall system speed. The number of ADAGs that are mapped to an architecture, $n_{ADAG}$, determines how many packets are processed during any given stage time. With a system clock rate of $clk$, the throughput of the system can be expressed as

$$throughput = n_{ADAG} \cdot \frac{clk}{\tau_{stage_{max}}}, \tag{1}$$

where

$$\tau_{stage_{max}} = \max_{i=1}^{D} \tau_{stage_i}. \tag{2}$$

The maximum stage time is the maximum time that any of the $D$ stages spends on processing, memory accesses, and communication.

## 5.2 Stage Time

The time that is required for a stage depends on the maximum time that any processing element spends for processing, accessing memory, and communicating. The processing time is $\tau_{proc_{i,j}}$ with $1 \leq i \leq W$ and $1 \leq j \leq D$ for processing element $i$ in stage $j$. Each processing element spends $\tau_{mem_{i,j}}$ for memory accesses, which depends on the contention on the memory interface.. The communication time, $\tau_{comm_i}$, is determined by how much data has to be transferred between stages. Thus the total stage time can be expressed as

$$\tau_{stage_j} = \tau_{comm_j} + \max_{i=1}^{W} (\tau_{proc_{i,j}} + \tau_{mem_{i,j}}). \tag{3}$$

In network processors, most processing elements are RISC-style processing cores. For such processors it can be assumed that one instruction is executed per clock cycles. Thus the processing time depends solely on the number of instructions, $instr_{i,j}$ that are mapped to this processing element during the mapping phase:

$$\tau_{proc_{i,j}} = \frac{instr_{i,j}}{clk}. \tag{4}$$

Modeling the communication time and memory access time are more challenging since these components are shared among multiple processing elements.

## 5.3 Memory Access Time

The memory access time is the sum of the queuing time due to contention and the actual memory access time. The queuing system can be modelled by a Machine Repairman model with a fixed number of sources (i.e., processing elements) and a certain number of servers (memory channels). When a memory access is issued, the processing element stalls until the request has been served. A mean value analysis of this queuing model can be found in work by Reijns and van Gemund [27]. We adopt the notation introduced by them.

To determine the memory access time we derive the system response time, $R_N$. $N$ is the number of processing elements that share memory interfaces and $M$ is the number of memory interfaces shared by those processing elements. The mean time between memory requests, $Z_i$, depends on the proportion of memory accesses (i.e., load and store instructions) to the number of non-I/O instructions. Each processing element requires $r_{i,j}$ read and $w_{i,j}$ write operations from/to the memory interface (given by the ADAG). $Z_i$ is calculated over the entire stage $i$ because memory channels are shared:

$$Z_i = \frac{\sum_{j=1}^{W} instr_{i,j}}{\sum_{j=1}^{W} (r_{i,j} + w_{i,j})}. \tag{5}$$

There is no closed form solution for the Machine Repairman system. Instead the result is computed incrementally over the number of job servers $N$. The response time for a system with $n$ job servers is [27]:

$$R_n = S + SQ_{n-1} - \frac{S}{2}U_{n-1}[1 - U_{n-1}][\frac{2}{S}R_{n-1} - 1], \tag{6}$$

where $S$ is the service time of the server (i.e., memory access time), $U$ is the utilization of the server, and $Q$ is the number request present at the memory interface (including the current one served). $U$ is defined as

$$U_{n-1} = \frac{SX_{n-1}}{M} \tag{7}$$

and $Q$ as

$$Q_{n-1} = \frac{X_{n-1}R_{n-1}}{M}, \tag{8}$$

with $X$ being the system throughput. $X$ depends on the response time of the system and the mean time between requests:

$$X_{n-1} = \frac{n-1}{R_{n-1} + Z}. \tag{9}$$

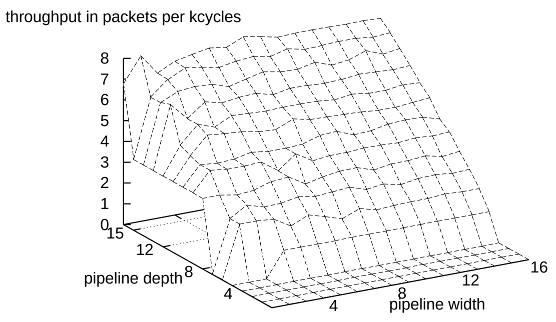
This allows a recursive calculation of $R_n$. The initialization values are $R_1 = S$ and $Q_0 = 0$. With the response time $R_N$, the overall memory access time for a processing element can be determined:

$$\tau_{mem_{i,j}} = R_n \cdot (r_{i,j} + w_{i,j}). \tag{10}$$

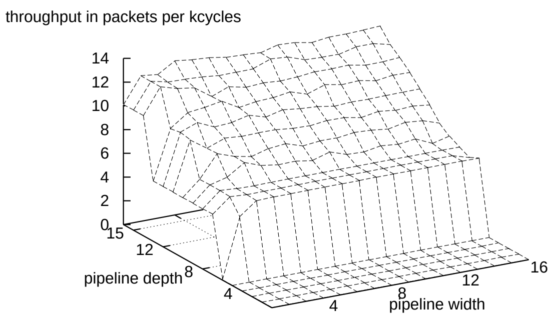
This model can be used for a range of memory technologies. By adapting the service time $S$, different memory technologies can be modelled. E.g., $S = 1$ could model an on-chip cache, $S = 10$ off-chip SRAM, and $S = 100$ off-chip DRAM.

## 5.4 Communication Time

The communication time of a stage depends on the total amount of data that needs to be transferred across the interconnect. The mapping of the ADAG yields information about data dependencies between different processing elements (indicated by arrows in Figure **??**. The amount of data that needs to be communicated between processing element $(i, j)$ and $(i', j')$ depends on the weight of the edge (shown in Figure 5) and is defined as $dep_{(i,j),(i',j')}$. Since the only means of communication between processing elements is the interconnect, the



(a) Flow Classification



(b) IPv4-trie

Figure 7: Throughput for Different System Topologies. The number of memory interfaces per stage is $M = 1$ and the memory service time is $S = 10$.

dependent data might need to traverse several interconnects to reach the next node in the mapped ADAG. Thus the communication delay depends on the number of dependencies between *all* prior stages and *all* later stages. The amount of data that is communicated across an interconnect $1 \leq k \leq D$, $data_k$, is:

$$data_k = \sum_{i \leq k, 1 \leq j \leq W} (dep_{(i,j),(i',j')} | i' > k \wedge 1 \leq j' \leq W) \quad (11)$$

To simplify analysis, we assume each data element can be communicated in one clock cycle. Thus the stage communication cost is:

$$\tau_{comm_i} = \frac{data_i}{clk}. \quad (12)$$

Equations 4, 10, and 12 can be substituted into Equation 3 to obtain the overall system throughput.

## 6 Results

This section presents and discusses the optimization results and performance tradeoff for various NPs topology parameters. Unless specified otherwise, the number of stages per interconnect is $I = 1$ and the number of randomized mapping runs is 100.

### 6.1 System Throughput

Figure 2 has already shown the impact of topology choices on system performance. There are a large number of configurations that perform poorly compared to the optimal solution for a given number of processors. To explore this issue in more detail, Figure 7 shows the system throughput for two of the four applications. The x- and y-axes vary the topology width, $W$, and depth, $D$.

Several observations can be made:

- There are several configurations, where the throughput is zero. This can happen for topologies where the ADAG cannot be mapped at all.

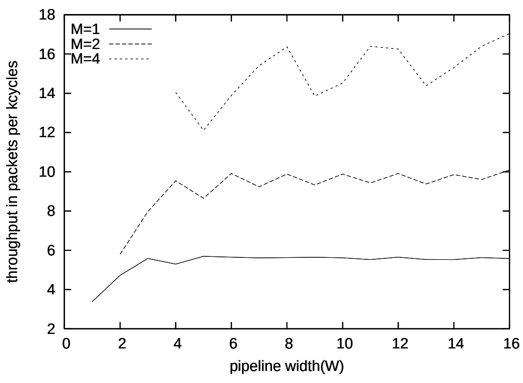


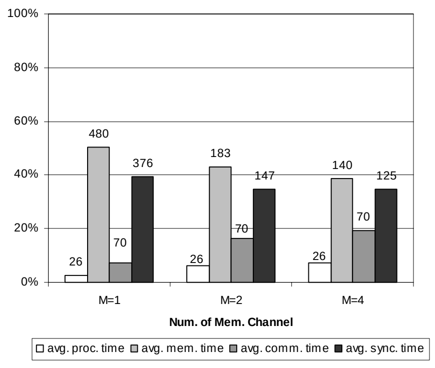
Figure 8: Throughput as a Function of System Width and Number of Memory Interfaces.

- The system throughput increases with the pipeline depth $D$. This increase is roughly proportional to the number of processors, which indicates that this dimension provides good scalability.
- For increasing pipeline width $W$, the throughput increases initially, but tails off for large $W$. This is due to the contention on the memory channel and severely limits scalability in this dimension.

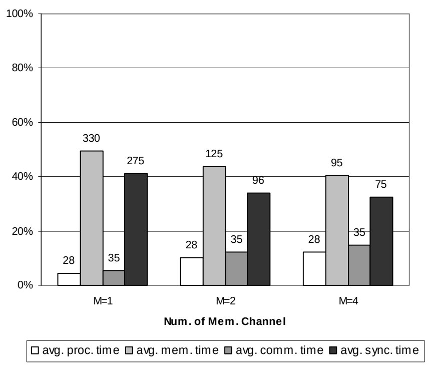
To further explore the impact of memory contention on the system performance and scalability, we compare the throughput for different memory interface configurations in Figure 8. By increasing the number of memory interfaces, the overall throughput can be increased almost proportionally. In all cases, the memory interface saturates with about 2 processing elements per interface.

### 6.2 System Bottlenecks

The above results indicate that memory contention can become a severe system bottleneck. To further illustrate how the total stage time is distributed between processing, memory access, and communication, Figures 9 and 10 show the contribution percentages. The "synchronization time," which is shown in these figures is the amount

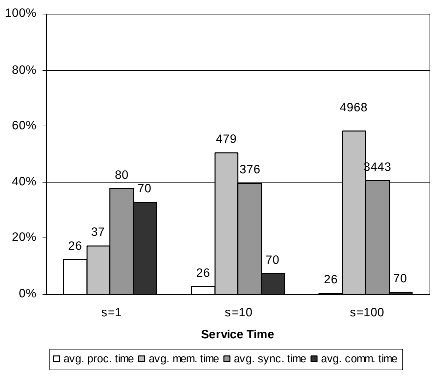


(a) Flow Classification

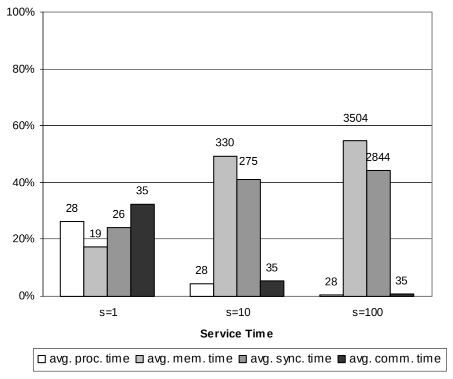


(b) IPv4-trie

Figure 9: Stage Time Distribution with Varying Number of Memory Channels. The number on top of the bars indicates absolute time in clock cycles.



(a) Flow Classification



(b) IPv4-trie

Figure 10: Stage Time Distribution with Varying Service Time. The number on top of the bars indicates absolute time in clock cycles.

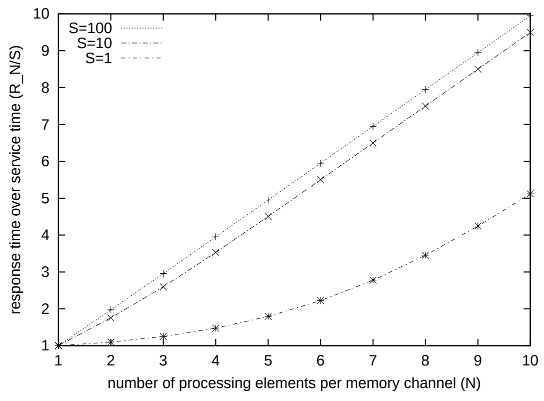


Figure 11: Memory Response Time as Fraction of Service Time over Different Number of Processors per Memory Interface.

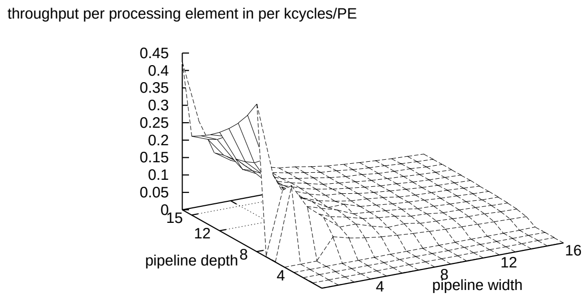
of time that a processing element has to wait due to the global pipeline synchronization. If a processing element finishes before the slowest processing element, it cannot advance but has to stall.

Figure 9 shows the system response time distribution for for different numbers of memory channels $M$. Here, the memory service time $S$ equals 10 clock cycles. Increasing $M$ reduces the memory access time, but even for $M = 4$, the memory access time dominates the overall stage time.

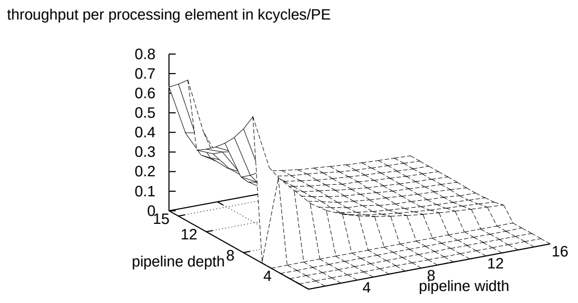
Figure 10 shows the system response time distribution when varying the memory service time $S$. Here, the memory channel per pipeline stage is $M = 1$. Only for a very small memory service time ($S = 1$), the memory access delay is comparable to communication and synchronization. This figure also shows that processing is *not* the limiting factor for throughput.

Another way of illustrating the impact of memory delay is shown in Figure 11. This figure shows the response time as a multiple of the service time over the number of processors that share a memory channel. The different lines show different memory technologies ($S = 100$ to $S = 1$). The observation is that for fast memories ($S = 1$) and a small number of processing elements, the response time is close to 1. This means that there is almost no queuing delay in the system and the memory access time is determined by the service time. For slower memories and larger numbers of processing elements, the response time is dominated by the queuing time, since it increases linearly with the number of processors. This means that the memory access time is not only larger due to the increased service time ($S = 10$ or $S = 100$) but also due to large queuing times (multiple service times).

All this leads to the conclusion that efficient memory systems are crucial for network processor topologies that



(a) Flow Classification



(b) IPv4-trie

Figure 12: Throughput per Processing Element Depending on Topology. The number of memory interfaces per stage is $M = 1$ and the memory service time is $S = 10$.

aim at achieving high throughput by exploiting parallelism through multiprocessing.

## 6.3 Design Tradeoffs

Another important question is how the overall system cost increases in order to achieve higher throughput. To illustrate this trade-off in a simple way without making any assumptions about implementation costs, we compare the throughput with the number of processing elements used for achieving this throughput.

Figure 12 shows the system throughput divided by the number of processing elements used over the different topology choices. For small values of the pipeline width $W$ the graph shows high values along the pipeline depth axis. This shows good scalability in the dimension of topology depth. The peaks in this figure coincide with mappings where ADAGs can be mapped without any idle processing elements. Along the pipeline width dimension, scalability is very limited due to performance decreases from memory contention.

The above results of our design space exploration can be used to extract a few general design guidelines for choosing network processor topologies:

- The network processor topology has significant impact on the overall system performance.

- Most network processing applications are of sequential nature (rather than inherently parallel). Exploiting this property leads to highly pipelined NP topologies. The task clustering and mapping process needs to aim at balancing the processing task to avoid slow pipeline stages.

- Memory access is a fundamental bottleneck. Large topologies need to be matched with a large number of high-speed memories to avoid these bottlenecks.

- Off-chip memory accesses cause a significant reduction in throughput and a drastic increase in queuing delay, which emphasizes the importance of latency-hiding techniques (e.g., multithreading).

- Next to the memory access delay, communication and synchronization are the main contributors to the pipeline stage time. Processing is not the limiting factor in the system throughput.

The performance model can be used to quantify these observations for any given topology and workload.

## 7 Conclusion

In this work, we have introduced a methodology to explore the network processor design space through performance modelling. The process involves abstracting NP workloads to annotated DAGs, which can be mapped to the general NP system topology. The performance model is integrated into the randomized mapping process to obtain a good approximation to the NP-complete mapping problem. The performance model takes the main system components into account and considers processing, memory accesses, inter-processor communication, and the effects of pipeline synchronization. We present results for a range of NP applications and topologies. The performance tradeoffs between different system topologies are presented and discussed. The general design guidelines that we derive from this work present the first step towards systematically evaluating NP topologies.

There are several extensions to this work that we are considering. First, the current performance model only considers general purpose processors. Most network processors use various types of hardware accelerators and co-processors. Such processing engines could be considered in our model to determine the proportions between general purpose and specialized processors for a given topology. Second, the mapping algorithm currently only allows one processing task to be mapped to a processing element. This leads to the side effect that some ADAGs cannot be mapped to some NP topologies (e.g., if there are fewer processors than nodes). We are currently expanding the mapping algorithm to allow for multiple nodes per processor and hope to present these results at the Anchor workshop. Third, the model does not yet consider multithreading. Multithreading has been shown to be a powerful powerful mechanism to hide the impact of off-chip memory access latency. Many modern NP architectures use multithreading and we intend to expand our model to take this into consideration.

We believe that the performance model that we provide in this work is a valuable tool for fast, quantitative design space exploration of network processor system topologies. This can help in determining how future NP architectures can be designed to provide the necessary scalability to support increasing link rates and increasing application complexity.

## Acknowledgements

The authors would like to thank Ramaswamy Ramaswamy for providing the run-time instruction traces that were used for the ADAG generation.

## References

[1] F. Baker. Requirements for IP version 4 routers. RFC 1812, Network Working Group, June 1995.

[2] D. P. Bhandarkar. Analysis of memory interference in multiprocessors. *IEEE Trans. on Computers*, c-24(9):897–908, Sept. 1975.

[3] D. Burger and T. Austin. The SimpleScalar tool set version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[4] P. Crowley and J.-L. Baer. A modelling framework for network processor systems. In *proc. of Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, pages 86–96, Cambridge, MA, Feb. 2002.

[5] P. Crowley, M. E. Fiuczynski, J.-L. Baer, and B. N. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proc. of 2000 International Conference on Supercomputing*, pages 54–65, Santa Fe, NM, May 2000.

[6] J. Daemen and V. Rijmen. The block cipher Rijndael. In *Lecture Notes in Computer Science*, volume 1820, pages 288–296. Springer-Verlag, 2000.

[7] EZchip Technologies Ltd., Yokneam, Israel. *NP-1 10-Gigabit 7-Layer Network Processor*, 2002. http://www.ezchip.com/html/pr_np-1.html.

[8] M. A. Franklin and T. Wolf. A network processor performance and design model with benchmark parameterization. In P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, editors, *Network Processor Design: Issues and Practices, Volume 1*, chapter 6, pages 117–138. Morgan Kaufmann Publishers, Oct. 2002.

[9] M. A. Franklin and T. Wolf. Power considerations in network processor design. In M. A. Franklin, P. Crowley, H. Hadimioglu, and P. Z. Onufryk, editors, *Network Processor Design: Issues and Practices, Volume 2*, chapter 3. Morgan Kaufmann Publishers, Nov. 2003.

[10] P. A. Grasso et al. Memory interference in multimicroprocessor systems with a time-shared bus. *Proceedings of the IEEE*, 131(10), Mar. 1984.

[11] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer. Exploring tradeoffs in performance and programmability of processing element topologies for network processors. In *Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Anaheim, CA, Feb. 2003.

[12] C. H. Hoogendoorn. A general model for memory interference in multiprocessors. *IEEE Trans. on Computers*, c-26(10):998–1005, Oct. 1977.

[13] Intel Corp. *Intel IXP1200 Network Processor*, 2000. http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[14] Intel Corp. *Intel Second Generation Network Processor*, 2002. http://www.intel.com/design/network/products/npfamily/ixp2400.htm.

[15] R. M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34(1-3):165–201, Nov. 1991.

[16] Y.-K. Kwok and I. Ahmad. FASTEST: A practical low-complexity algorithm for compile-time assignment of parallel programs to multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 10(2):147–159, Feb. 1999.

[17] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, Dec. 1999.

[18] V. Lakamraju, I. Korean, and C. M. Krishna. Filtering random networks to synthesize interconnection networks with multiple objectives. *IEEE Trans. Parallel Distributed Systems*, 13(11):1139–1149, Nov. 2002.

[19] Lucent Technologies Inc. *PayloadPlus$^{TM}$ Fast Pattern Processor*, Apr. 2000. http://www.agere.com/support/non-nda/docs/FPPProductBrief.pdf.

[20] B. A. Malloy, E. L. Lloyd, and M. L. Souffa. Scheduling DAG's for asynchronous multiprocessor execution. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):498–508, May 1994.

[21] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A benchmarking suite for network processors. In *Proc. of International Conference on Computer-Aided Design*, pages 39–42, San Jose, CA, Nov. 2001.

[22] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[23] National Laboratory for Applied Network Research - Passive Measurement and Analysis. *Passive Measurement and Analysis*, 2003. http://pma.nlanr.net/PMA/.

[24] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, June 1999.

[25] R. Ramaswamy, N. Weng, and T. Wolf. Application analysis and resource mapping for heterogeneous network processor architectures. In *Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-10)*, pages 103–119, Madrid, Spain, Feb. 2004.

[26] R. Ramaswamy and T. Wolf. PacketBench: A tool for workload characterization of network processing. In *Proc. of IEEE 6th Annual Workshop on Workload Characterization (WWC-6)*, pages 42–50, Austin, TX, Oct. 2003.

[27] G. L. Reijns and A. J. C. van Gemund. Analysis of a shared-memory multiprocessor via a novel queuing model. *Journal of Systems Architecture*, 45(14):1189–1193, 1999.

[28] TCPDUMP Repository. *http://www.tcpdump.org*, 2003.

[29] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In *Proc. of Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, pages 30–41, Cambridge, MA, Feb. 2002.

[30] A. J. C. van Gemund. Performances prediction of parallel processing systems: The pamela methodology. In *Proc. 7th ACM International Conference on Supercomputing*, pages 318–327, Tokyo, Japan, July 1993.

[31] Y.-C. Wei and C.-K. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(7):911–921, July 1991.

[32] T. Wolf and M. A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX, Apr. 2000.