International Journal of Parallel Programming, Vol. 35, No. 1, February 2007 (© 2006)
DOI: 10.1007/s10766-006-0023-0

FEADS: A Framework for Exploring the Application Design Space on Network Processors

Rajani Pai[1] and R. Govindarajan[1,2]

Received November 23, 2005; accepted September 7, 2006

Network processors are designed to handle the inherently parallel nature of network processing applications. However, partitioning and scheduling of application tasks and data allocation to reduce memory contention remain as major challenges in realizing the full performance potential of a given network processor. The large variety of processor architectures in use and the increasing complexity of network applications further aggravate the problem. This work proposes a novel framework, called FEADS, for automating the task of application partitioning and scheduling for network processors. FEADS uses the simulated annealing approach to perform design space exploration of application mapping onto processor resources. Further, it uses cyclic and r-periodic scheduling to achieve higher throughput schedules. To evaluate dynamic performance metrics such as throughput and resource utilization under realistic workloads, FEADS automatically generates a Petri net (PN) which models the application, architectural resources, mapping and the constructed schedule and their interaction. The throughput obtained by schedules constructed by FEADS is comparable to that obtained by manual scheduling for linear task flow graphs; for more complicated task graphs, FEADS' schedules have a throughput which is upto 2.5 times higher compared to the manual schedules. Further, static scheduling of tasks results in an increase in throughput by upto 30% compared to an implementation of the same mapping without task scheduling.

KEY WORDS: Cyclic scheduling; design space exploration; network processor; programming model; performance Evaluation; petri Nets.

[1]Department of Computer Science and Automation, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560 012, India.
[2]To whom correspondence should be addressed. E-mail:{rajani, govind}@csa.iisc.ernet.in

1. INTRODUCTION

Network processors provide a cost-efficient yet flexible solution for developing applications that process packets at high data rates. With increasing network traffic, the demand for higher performance, typically specified in terms of line rates, is on the increase. Network devices of today need to handle OC-48 lines rate or higher, where OC-48 (Optical Carrier 48) is a fiber optic network line with a SONET rate of 2488.32 Mbit/s or 48 times the basic SONET signal transmission rate. The performance requirement of network processors depends on where in the network they are being used. While applications at the edge of the network are expected to handle low data rates with more processing done per packet, those at the core need to perform a small set of simple tasks at high data rates.

Network processing applications consist of a single loop for packet processing. Most packets in network traffic do not exhibit interdependencies and can therefore be processed independently. Network processors are designed to exploit this inherently parallel nature of applications. Most architectures consist of multiple processing elements connected in a pipelined(16) or multiprocessing fashion,(8–11) with communication between processors taking place either through memory or through special-purpose registers.(9–11) Each processing element supports multithreading, which hides large memory latencies by allowing multiple outstanding memory accesses from a processing element. Also, dedicated hardware may be available for compute-intensive tasks like encryption and checksum computation. Different types of memory are used, such as DRAMs for storing received packets, SRAMs for storing routing tables, and various other lower capacity memory elements for communication between processing elements. Not only do architectural features vary widely across processor families, but differences also exist within each processor family thus exacerbating portability issues. A generic network processor architecture is shown in Fig. 1.

It is the responsibility of the programer to assign processing tasks and data to the available resources as well as to schedule these tasks so as to meet the performance requirements of the application. It has been observed that the critical factors in programing network processors are partitioning of tasks onto processors and threads, scheduling of resources, assignment of data to memory elements, and data transfer management.(13) Additionally, other constraints like size of the instruction store and capacities of the memory elements need to be considered during binding of tasks to resources. Further, scheduling multiple instances of the

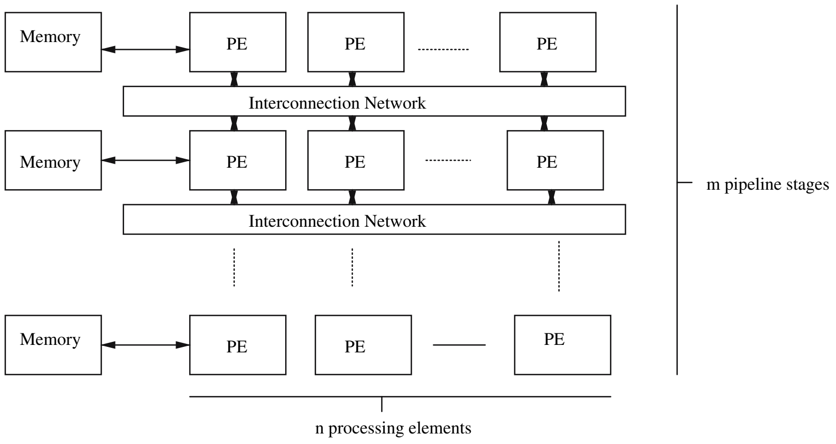


Fig. 1. Generic network processor architecture.

application might yield better throughput than a single instance as will be shown in Section 4.2.2.

This is because the initiation interval for the schedule has to be an integral value, and if the minimum initiation interval computed for a given schedule is fractional, then it is rounded up to the next integral value. Thus considering n instances, where n is chosen such that the computed minimum initiation interval is an integral value, may improve the overall throughput. This is similar to the r-periodic schedule discussed in Ref. 20. Hence the number of instances required to arrive at the optimal throughput needs to be determined.

The above functions, generically referred to as mapping and scheduling of tasks in this work, become increasingly more complex as larger network applications are mapped onto newer and different processor families. It is imperative to automate this process to explore the design space and choose the most suitable mapping of an application to the given architecture and find an optimal schedule for the mapping. However, the mapping and scheduling process can only provide an estimate for (packet) throughputs, which are static in nature, under constant packet arrival assumption. To get a better picture of the performance of scheduling and mapping schemes, it is necessary to obtain *dynamic performance metrics*, such as packet throughput and utilization of various resources (e.g., micro-engines, DRAM, and SRAM) under realistic workloads, such as packet traffic modeled using Poisson distribution. Such dynamic performance metrics also reveal bottleneck resources and insights for remapping and scheduling. Hence the performance results obtained from the performance module can be fed back to the scheduler module to further tune the scheduling and mapping process. In order to accomplish this, it is necessary to automate this process of obtaining dynamic performance metrics as well.

In this paper, we propose FEADS, a framework for exploring the application design space on network processors, which can evaluate various alternatives in application partitioning, scheduling and mapping onto the network processor architecture and choose the best alternative. We address these issues by partitioning the application into tasks based on whether they utilize the processor or memory, thus yielding tasks of finer granularity. The application is represented as an Annotated Directed Acyclic Graph (ADAG) where nodes represent the fine-grain tasks performed on a packet. For example, a node in an ADAG may represent the *lookup* task performed by packet forwarding applications.

The design space of mapping the tasks to resources is explored using simulated annealing.(12) The mapping scheme can take into account various constraints such as code store size, communication options, and their latencies. It explores whether pipelined or parallel, or a combination of these mappings is better. Once a mapping is finalized, the tasks are scheduled using cyclic scheduling method (software pipelining). We use a variant of decomposed software pipelining method.(21) Further, we consider unrolling of the task flow graph to obtain better throughput. This is similar to r-periodic scheduling.(20) We propose a simple heuristic to estimate the number of instances required. To obtain the dynamic performance metrics, we use the Petri net (PN) model developed in Ref.(6) for performance evaluation of network applications on network processors. A salient feature of the PN model is that it models the application, the architecture, the mapping and schedule derived by FEADS, and their interaction in detail. The PN model is automatically generated by our PN_Generate tool. The PN generated changes whenever the architecture, application, schedule or the mapping changes. The generated PN is simulated using CNET,(23) a PN simulator tool, and performance metrics are obtained.

The throughput obtained by the mapping generated by FEADS for the applications under consideration — IPv4, NAT, and IPSec — is comparable to that obtained by a manual binding of tasks to resources as in Ref.(6). Further, enforcing static scheduling of tasks, as opposed to scheduling packets immediately upon arrival, results in an increase in throughput by 30%. In the case of IPSec, with code size constraints imposed on the mapping, the mapping and schedule produced by FEADS results in 27% higher throughput than the manual mapping. Further studies with more complicated task graphs showed that our framework yields upto 2.5 times higher throughput as compared to a manual binding under code size constraints. This clearly indicates that with the growing complexity of network applications and increasing resource constraints, manual binding cannot adequately meet the line speeds required by these applications.

The rest of the paper is organized as follows. First, a discussion on related work is presented. In Section 3, a brief description of various processors in the IXP family of network processors is presented. Section 4 presents FE-ADS, a framework which performs task partitioning and scheduling. Section 5 presents the experimental results. Finally, Section 6 concludes the paper.

2. RELATED WORK

Kulkarni *et al.*, studied the performance impact of partitioning on different network processor architectures.(13) An IP forwarding router application was implemented on the IXP1200 and Motorola C-5 network processors. The application was partitioned at the logical cutting points revealed by the application, such as packet receive, header processing, and packet transmit. It was observed that inefficient partitioning negatively impacted the throughput by more than 30%, and localization of computation related to the memories increased the available bandwidth on internal buses by a factor of two.

Several methodologies for the design space exploration of network processors have also been proposed. Specifically, Blickle *et al.*,(1) and Thiele *et al.*,(19) use a genetic algorithm to explore the design space of network processors. Given an application, flow characterization and available resources, they bind and schedule the tasks of the application onto the resources. The objective function considers the cost of implementing the given mapping in hardware. Once a set of optimal points are obtained, simulation can be performed to select the best architectural configuration for the given application and flow characterization. Their work looks at the problem from an architectural design perspective, as reflected by their objective function, while we approach it from an application design viewpoint. Moreover, common features of network processors like the availability of multiple threads on a processing engine are not considered by them while binding tasks to resources. They also do not accommodate the fact that some tasks can be bound to more than one resource at a given time, such as a processor thread and memory element, as is the case with memory-bound tasks. Further, they have used an evolutionary approach to explore the solution space, while we use simulated annealing since we found that it covers the solution space more effectively and arrives at a better solution.

Weng and Wolf(22) construct an ADAG of the application using runtime traces and use this to perform design space exploration. An

ADAG is a directed acyclic graph whose nodes are annotated with their processing and memory requirements, while the edges are annotated with the number of bytes of communication between the nodes connected by the edge. They use a randomized algorithm to perform mapping of nodes to processors and memories. 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 pipeline stages.Memory contention is modeled using a queuing network approach. At the end of the mapping process, the best overall mapping is reported. It should be noted that their model considers a network processor architecture with fixed binding of processors to memory and communication elements, and hence does not take into account the availability of multiple memory modules for binding even after the processor has been selected. Since we use a simulated annealing approach, we believe that our framework will explore the design space more thoroughly as compared to multiple randomized mappings, where each solution point is independent of the other.

Franklin and Datar(4) propose an algorithm which employs a greedy heuristic to schedule tasks derived from multiple application flows on pipelines with an arbitrary number of stages. Tasks may be shared, and different bandwidths may be associated with each of the application flows. However, memory contention is not modeled. They have observed that task partitioning results in greater flexibility in assigning tasks to the hardware pipeline, but with a corresponding increase in the complexity of implementing the partitioning and larger inter-task communication costs. Sharing of tasks common to different flows also leads to better throughput, since memory contention is reduced. The task graph is partitioned at arbitrary points to achieve load balancing across processors, while in our work, we start with a more clearly demarcated set of processing, memory and communication tasks. The demarcation of task boundaries with memory and communication task enables us to easily take into account the memory and communication costs and possible thread context switching.

The Shangri-la system proposed by Chen *et al.*,(2) consists of a programing language, a compiler for optimizing network programs using both traditional and specialized optimization techniques, and a runtime system which identifies hot code paths and improves their mapping to processing resources. Aggregation of tasks from different pipe stages is done so as to minimize communication cost. Pipeline aggregate duplication is used to improve the throughput of the slowest pipe aggregate if its throughput is much less than the other pipeline aggregates. We have found that scheduling multiple instances of the application achieves the same result as duplication of individual pipe stages.

Click[17] and NP-Click[18] provide a programing abstraction to specify network applications. Click is a software architecture comprising of packet processing modules called elements. To build a router configuration, the user connects a collection of elements into a graph. Packets move from element to element along the graph's edges. NP-Click[18] builds on this framework to provide an abstraction of the underlying hardware and data layout that exposes enough architectural detail to write efficient code for that platform, while hiding less essential architectural complexity. In either case, elements have to be rewritten for portability. Also, the tasks under consideration involve many memory accesses internally and hence it is difficult to model contention for memory. We have used an ADAG with tasks of finer granularity which can be generated from an application specified using either of these abstractions.

Ennals *et al.*,[3] describe transformations that can be performed on network processing applications such as pipelining tasks and merging pipelined tasks. Since their work mainly deals with task partitioning, it is orthogonal to our work.

3. IXP FAMILY OF NETWORK PROCESSORS

Figure 2 shows a block diagram of the Intel 2400 network processor,[9] which consists of an Intel XScale core and eight microengines. The Intel XScale core initializes and manages the chip and is used for control plane functions like exception handling. The microengines (MEs) are 32-bit programable processors specialized for network processing, which handle the data plane processing for the packets. Each ME supports eight threads and allows zero-overhead context switching.

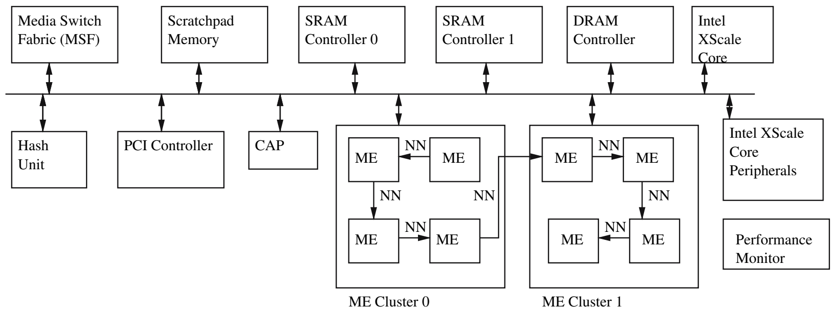


Fig. 2. Block diagram of IXP2400.[9]

The memory architecture of the IXP2400 consists of SRAM, DRAM, scratch memory, local memory, and next neighbor registers. Typically, packets are stored in DRAM memory, while the SRAM memory stores route lookup tables. 16 KB of low-latency scratchpad memory is also provided, which along with SRAM memory can be used for communication between MEs. There are separate SRAM and DRAM controllers for interfacing the external memories. Next neighbor (NN) registers can be used for communication between adjacent MEs. Communication between NNs is unidirectional, as shown in Fig. 2. Additionally, each ME consists of 640 words of local memory, which can be used for communication between hardware contexts. Specialized hardware like the hash and crypto units are also available on processors in the IXP family.

As can be seen, there is a wide variety of options for mapping communication and processing tasks. Moreover, the resource used for communication and hence the latency is constrained by the processor onto which adjacent tasks in the task flow graph are bound. The size of the solution space under consideration implies that the function of mapping tasks to processors and memory modules should be automated.

Other processors in the IXP family include the IXP1200, the IXP2800, and the IXP2850. These processors have a similar structure, but the number of MEs, memory controllers and the specialized hardware differ. The IXP1200 consists of six MEs with four threads per ME.(8) It has only one SRAM controller, one DRAM controller and no next neighbor registers for communication. The IXP2800, on the other hand, has 16 MEs distributed over two clusters, four SRAM controllers and three DRAM controllers.(10) Other features, however, remain identical to the IXP2400. The IXP2850 is similar to the IXP2800, but additionally consists of two crypto units which implement cryptographic algorithms in hardware.(11)

Due to these dissimilarities in the architecture of processors in the IXP family, the binding done for one processor might not lead to an optimal solution for the others. Task partitioning and performance evaluation techniques therefore need to consider the various resources available on each processor in order to utilize them effectively.

4. FRAMEWORK FOR TASK SCHEDULING AND PARTITIONING

Figure 3 depicts the design of FEADS, a framework for task scheduling and performance evaluation for a given architectural specification. The details of the different components of the entire framework are described in detail below.

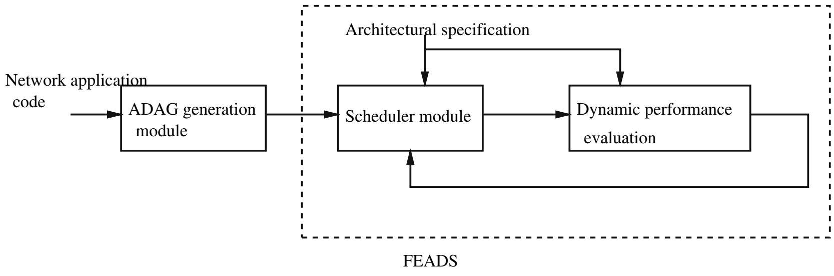


Fig. 3. Proposed design for the framework.

4.1. ADAG Generation Module

The application to be scheduled can be specified using a framework like Click or NP-Click. An ADAG for this task flow graph is generated, along with various constraints like resources that the tasks can be mapped onto and instruction store requirements of tasks. Task nodes in the ADAG are explicitly classified into processing, memory, communication, or application-specific tasks meant for the hash or crypto units. The nodes of the ADAG are considered to be atomic and cannot be split further. Based on the type of node, it is either annotated with the processing requirements in terms of the number of cycles or the memory requirements in terms of the number of bytes to be transferred. Memory nodes are also annotated with information about the type of memory — SRAM or DRAM — that they need to access. Similar constraints can also be imposed on the binding of other nodes. Since we use nodes of finer granularity than those provided by Click or NP-Click, we model the memory contention and communication costs with greater accuracy and hence obtain a better schedule.

An example ADAG is shown in Fig. 4. Processing nodes are denoted by P, memory nodes by M, and communication nodes by C. Communication nodes are annotated with the number of bytes of communication needed by the two adjacent nodes. For example, the communication from nodes a to b requires 32 bytes, and is represented by node e in Fig. 4. The annotations for node a indicate that it requires 10 clock cycles for execution and should be bound to a ME. Similarly, node b transfers 32 bytes of data and should be bound to SRAM (S). Node c should be bound to DRAM (D).

In this work, we do not concentrate on the ADAG generation module. We assume that an ADAG representation of the application forms an input to FEADS. For the example network applications considered in this paper, viz., IP forwarding (IPv4), Network Address Translation (NAT), and IP Security (IPsec) (refer to Section 5.1 for a detailed discussion on

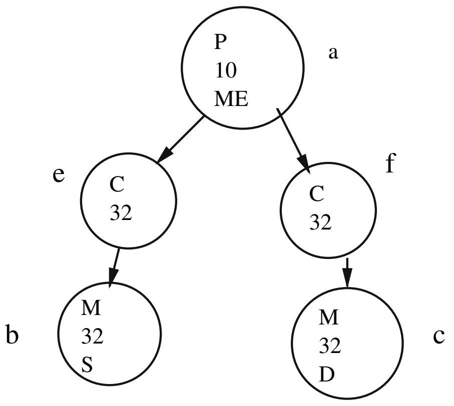


Fig. 4. Nodes in an ADAG.

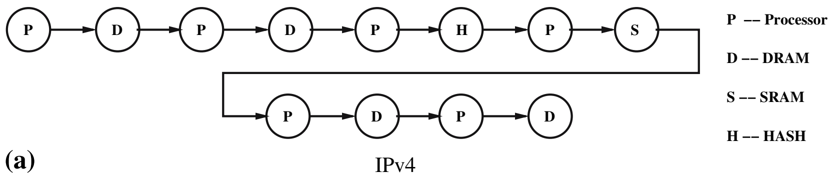
these applications), we have shown the ADAGs in Fig. 5. The generation of the ADAG for an application is left for future work.

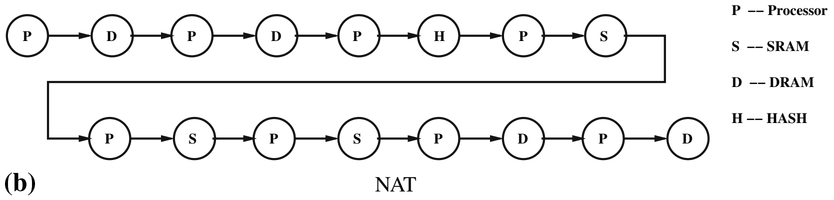
4.2. Scheduler Module

The ADAG and architectural specification serve as inputs to the scheduler, which performs mapping of ADAG nodes onto the appropriate resources. The architecture of the processor on which the application is to be scheduled is specified as a pair consisting of the type of resource and number of resources of that type available. The overall design and implementation of the scheduler module is shown in Fig. 6. The outer loop estimates the number of instances which would result in optimal per instance throughput and performs a mapping and scheduling of the estimated number of instances using simulated annealing. Details are discussed in Section 4.2.2. PN_Generate is a part of Dynamic Performance Evaluation module. The inner loop uses simulated annealing to generate a mapping for the given ADAG. For each mapping, the tasks are scheduled and the schedule length is obtained. We use schedule length as the static performance metric for comparing different mappings. The simulated annealing algorithm and the other modules are described in the following subsections.

4.2.1. Task Mapping

Simulated annealing[12] is used to explore the task mapping space. Mapping is done using Algorithm 1. The schedule length for a mapping is





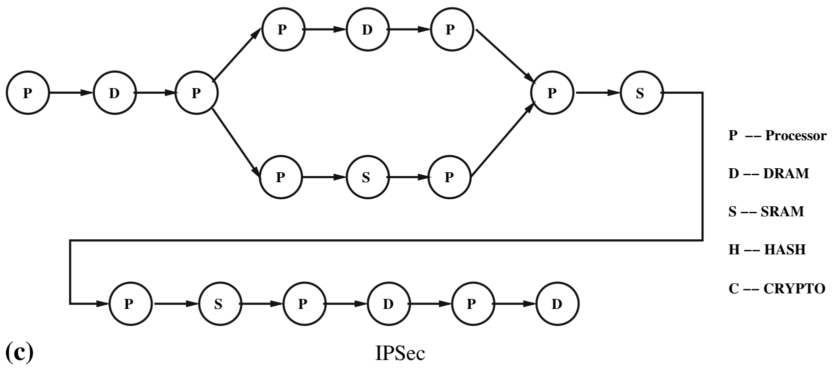


Fig. 5. ADAGs for IPv4, NAT, and IPSec.

used as the objective function. Starting with an initial binding, successive mappings for a given temperature are incrementally derived. The incremental change in the mapping for a task to a given resource is a function of the temperature under consideration. At higher temperatures, there can be larger changes in the binding of tasks to a resource in terms of the distance from the previous mapping. As the temperature decreases, the amount of perturbation allowed also reduces. When a mapping changes, some of the bindings previously associated with communication tasks might no longer be valid due to changes in the bindings of tasks which they connect and hence the mapping has to be repaired. For each mapping thus obtained, a schedule is constructed as described in Section 4.2.3. A mapping with lower schedule length is unconditionally selected for annealing, while mappings with higher schedule length are selected with probability

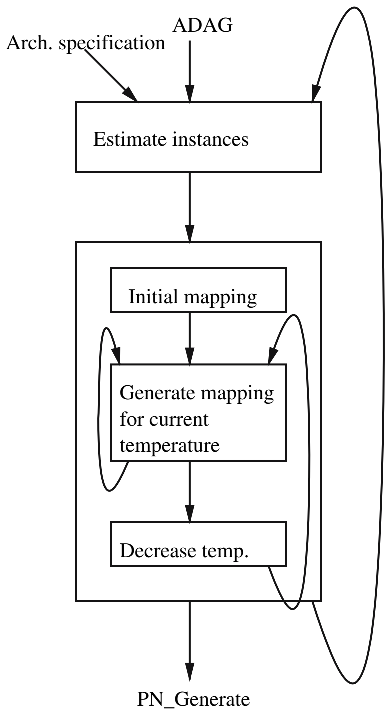


Fig. 6. Overall design of scheduler module.

$$\exp\left(\frac{-(\text{currSchedLen} - \text{prevSchedLen})}{k * T}\right),$$

where k is the Boltzmann constant and T the current temperature. After every iteration of the inner loop, the temperature is reduced and the process is repeated. Use of simulated annealing for exploring the design space is acceptable since the mapping and scheduling need to be done only once when the processor is programmed.

4.2.2. Scheduling Multiple Instances

Mapping a single instance of the application might result in some resources being idle. For example, for the ADAG shown in Fig. 7(a), a single instance resulted in a schedule shown in Fig. 7(b) with a throughput of 0.5. However unrolling the ADAG multiple times and cyclic scheduling the unrolled ADAG can result in higher throughput. Scheduling two and three instances of the ADAG results in an II (Initiation Interval) of

Algorithm 1 Algorithm for Mapping using Simulated Annealing

```
$prevBinding = Generateinitialbinding$
$prevSchedLen = schedule(prevBinding)$
**for** $T = initialTemp$ to $finalTemp$ **do**
  **for** $j = 1$ to $N$ **do**
    $newBinding = f(prevBinding, T)$
    $newSchedLen = schedule(newBinding)$
    **if** $newSchedLen < prevSchedLen$ or $random(0, 1) >$
    $\exp(-(currSchedLen - prevSchedLen)/(k * T))$ **then**
      $prevBinding = newBinding$
      $prevSchedLen = newSchedLen$
    **end if**
  **end for**
  T=T-1
**end for**
```

three and four, respectively, which corresponds to a throughput of $\frac{2}{3}$=0.67 and $\frac{3}{4}$=0.75 packets per cycle, respectively, as shown in Fig. 7(c) and (d). A further increase in the number of instances does not yield any improvement in throughput since all the PEs are saturated.

Replicating instances of the application is equivalent to loop unrolling and results in better resource utilization and per instance throughput. Since incoming packets are independent of each other, only resource dependencies need to be considered while scheduling instances of tasks across loops. The minimum initiation interval (MII), governed only by resource dependencies for k instances of the ADAG, is given by

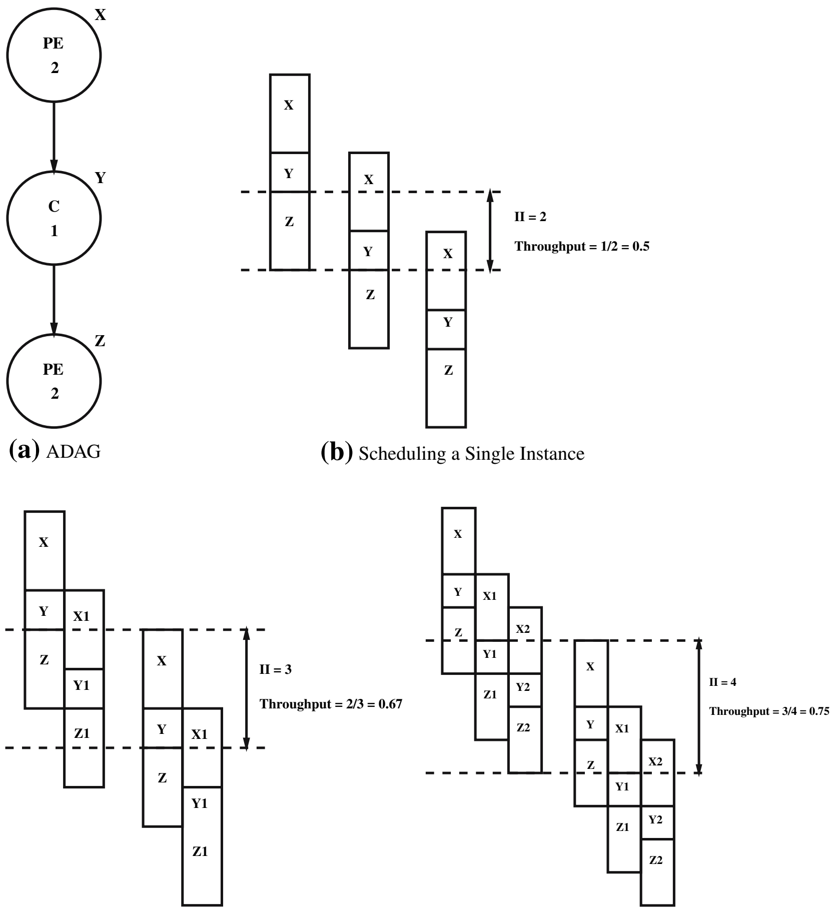
$$\mathrm{MII} = \max_{f}\left(\left\lceil \frac{i * nT_{\mathrm{f}} * lat_{\mathrm{f}}}{nR_{\mathrm{f}}} \right\rceil\right),$$

where f the resource type,

nT_{f} the number of tasks bound to resource f, and
lat$_{\mathrm{f}}$ the latency of resource f.

The throughput of a schedule with such II is $\frac{k}{\mathrm{II}}$ and the average II per instance is $\frac{\mathrm{II}}{k}$. Since we want to obtain maximum throughput with minimum amount of unrolling, the lower bound for the II of the unrolled and pipelined loop schedule is computed as

$$\min_{i}\left(\frac{\max_{\mathrm{f}}\left(\left\lceil \frac{i * nT_{\mathrm{f}} * lat_{\mathrm{f}}}{nR_{\mathrm{f}}} \right\rceil\right)}{i}\right), \qquad (1)$$



(a) ADAG **(b)** Scheduling a Single Instance

(c) Scheduling Two Instances **(d)** Scheduling Three Instances

Fig. 7. Scheduling Multiple Instances of an ADAG.

where i the number of instances,

f the resource type,
nT_f the number of tasks bound to resource f,
lat_f the latency of resource f, and
nR_f the number of resources of type f.

We use the concept of r-periodic scheduling(20) to determine the optimal number of application instances to be scheduled. In this scheme, the schedule of two consecutive iterations of a loop may not be the same, but will repeat every r iterations. In our case, r determines the number of instances of the application to be scheduled. When multiple instances of the application are to be scheduled, the ADAG is replicated and the nodes are mapped onto the resources and scheduled.

From the equation it can be seen that multiples of the resource with the largest occupancy yield the best theoretical lower bound and hence these can be used as starting points for further exploration in our design. It might not always be possible to obtain a schedule with an II equal to the computed lower bound because our mapping might result in some additional communication tasks being bound to some of these resources. For example, if tasks X and Y operate on some common data, but are bound to two different MEs, a communication task with additional latency is generated as part of the mapping. Also, since some tasks might be bound to more than one resource, the actual schedule length might differ from the computed one.

4.2.3. Task Scheduling

For each mapping, FEADS constructs a valid schedule. Since cyclic or software pipelined schedules(14) yield a better throughput than block schedules, FEADS considers only cyclic schedules. In cyclic scheduling, successive iterations of the ADAG are overlapped. A specific form of cyclic scheduling, modulo scheduling, schedules different instances of a node, corresponding to different iterations, exactly II cycles apart. The throughput achieved by such a schedule is $\frac{1}{II}$. Hence cyclic scheduling attempts to minimize the II. Consider the ADAG in Fig. 7(a) with the available resources being three processing engines (PE) and one memory element. Assume the PEs are non-pipelined. Nodes x and z require two cycles each for execution, while node y takes one cycle. Cyclic scheduling results in the schedule shown in Fig. 7(b). The throughput of this schedule is $\frac{1}{2}$ packets per cycle. Note that in this schedule instances corresponding to three iterations overlap in an II.

To construct an r-periodic cyclic schedule for a given II (computed using Eq. 1), and the mapping obtained using the simulated annealing. we use decomposed software pipelining algorithm.(21) We have used the First Row Last Column (FRLC) algorithm for decomposed software pipelining in our implementation. Here the row numbers denote the cycles of execution and the column numbers denote the iteration. Since there are no loop carried dependences in our graph, the set of Strongly Connected Components (SCC) is empty. We generate the new Loop Data Dependence Graph (LDDG) by removing all edges which do not connect the SCCs, as per the algorithm. We then perform list scheduling(7) on this newly generated LDDG. The row number for each task is determined by the cycle at which it starts execution. The II for the pipelined loop is given by the number of rows as computed by the FRLC algorithm.

4.3. Performance Evaluation Module

A dynamic performance evaluation technique is incorporated into FEADS to study the performance of the statically generated schedule under realistic workloads. A PN model of the application mapping and schedule on the given processor is generated for dynamic performance evaluation. This is similar to the technique used by Weng and Wolf(22) wherein a performance model of the architecture under consideration is used to compute the throughput due to a given binding using queuing-network approach.

A brief description of PN models in general and their use in modeling applications mapped onto network processors in particular is given in the following subsections. We also describe the scheme used by our framework to automate the generation of these models for use by the CNET simulator(23) for PNs.

4.3.1. The Petri Net Model

The PN is a mathematical modeling tool which is commonly used to model concurrency and conflicts in systems. Circles represent places, boxes represent the timed transitions. Timed Petri nets (TPN) are an extension to PN where a finite firing time is associated with transitions. The TPNs have been used in modeling multithreaded processors. The main advantage in using TPN for processor modeling is the added ability to capture the latency of operations like memory access and processor execution at a high level of abstraction.

The PN model generated by our framework models the architecture as well as the application in great detail. An example model of a single microengine in IXP2400 running the IPv4 application is shown in Fig. 8. For clarity only a part of the model which captures the flow of packets from the external link to DRAM through the MAC is shown. The firing time of a timed transition is assumed to be deterministic. The places UE, THREAD, UE_CMD_Q, DRAM_Q, CMD_BUS, represent various resources available and thus model the processor architecture, and the timed transitions UE_PROCESSING and RFIFO_DRAM represent the specific tasks. The time taken by these transitions model the time taken by these tasks in the specific unit. Thus the PN model is able to capture the processor architecture, applications and their interaction in detail. A

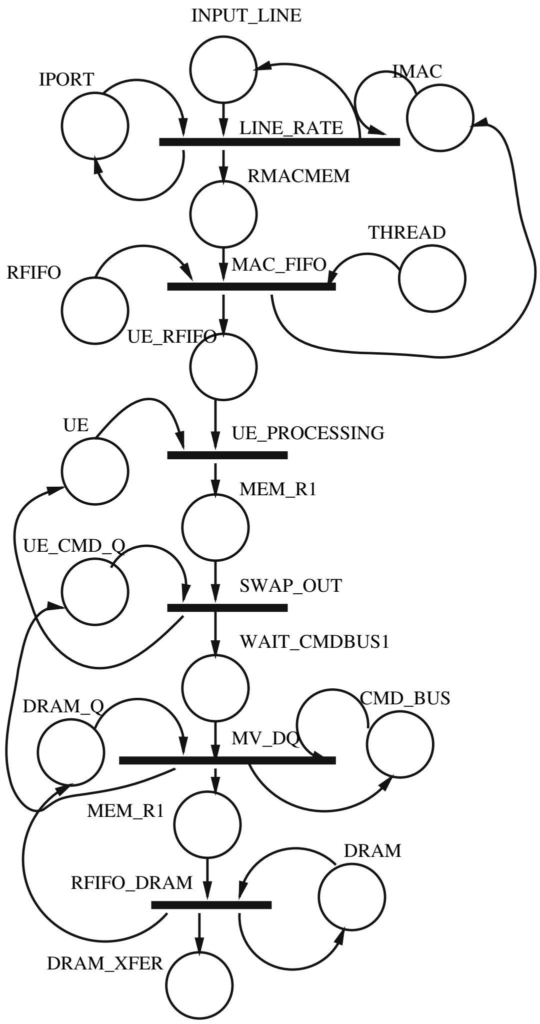
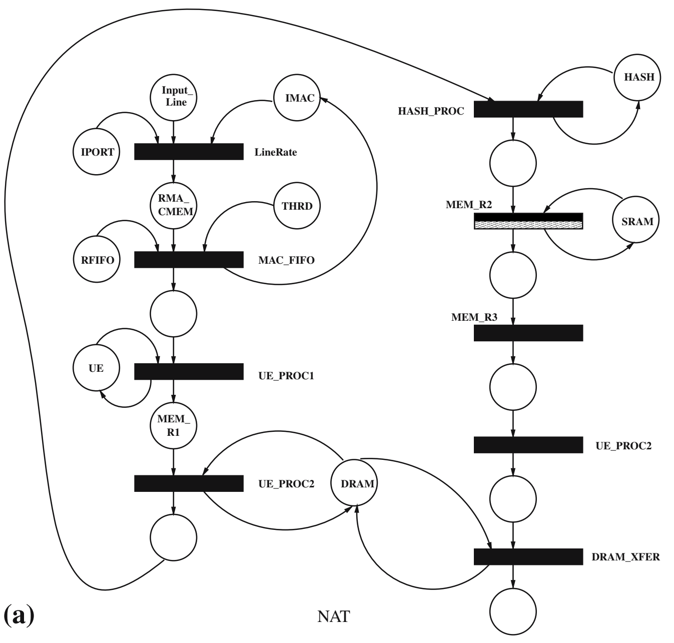


Fig. 8. PN model for a single microengine in IXP2400 running IPv4.

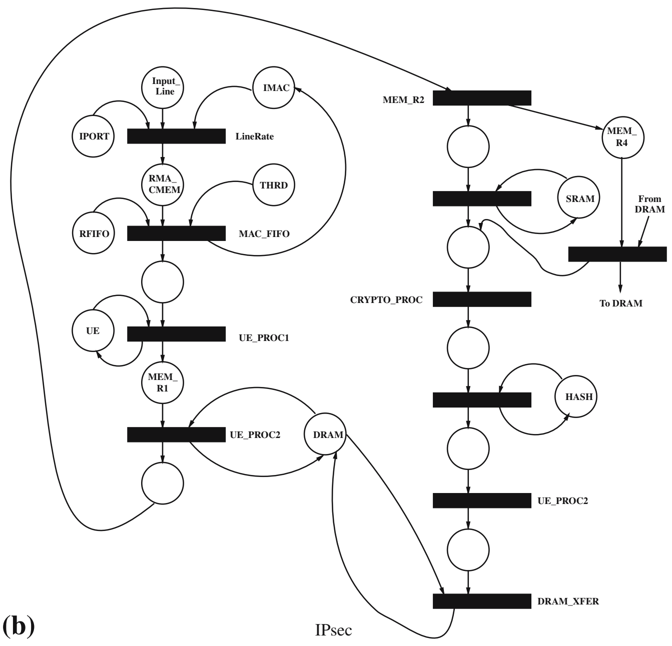
more detailed explanation of the model is given in Ref.[6] The PN models forNAT and IPSec are given in Fig. 9.

4.3.2. The PN_Generate Tool

The dynamic performance of the mapping and schedule generated by the scheduler module is evaluated using the PN model discussed in the



(a)



(b)

Fig. 9. PN models for a single ME in IXP2400 running NAT and IPSec.

previous subsection. This model is simulated using the CNET simulator for PNs.(23) We generate the PN model of the application mapping onto the processor and this forms the input to the CNET simulator. We traverse the ADAG and generate transitions for each node. Resources to which the task is bound are acquired during the beginning of a timed transition and released when the transition fires. Dummy transitions are generated between two places to pause between transitions and hence impose the computed static schedule on the model.

For the example shown in Fig. 8, if the task UE_PROCESSING takes 10 cycles for execution, its transition for the CNET simulator is generated as

$$\text{No.UE\_PROCESSING} : D * 10\{\text{UE}, \text{UE\_RFIFO/MEMR1}\},$$

where D indicates deterministic firing time. Terms to the left of the/correspond to input places and represent resources that the transition needs for execution and those to the right correspond to output places and represent resources released by the task upon completion. In generating the PN, the tool makes use of the mapping generated by the simulated annealing method, and an appropriate resource is used as an input place. Similarly, the number of instances and the cyclic schedule generated by our scheduling method is used by the PN_Generate tool.

The binding, scheduling, and verification stages have been made generic enough to allow FEADS to be used for application mapping on other processors in the IXP family. Only some small modifications to the architectural description file need to be made, specifying the change in the number and type of available resources.

5. RESULTS

In this section, first, we present our experimental setup, including a brief description of the application programs considered in our study. The subsequent sections discuss performance results of FEADS.

5.1. Experimental Setup

We evaluated the schedules generated by FEADS for three network processing applications — IPv4, IPSec, and NAT. A brief description of each application and the sizes of their ADAGs are given in below.
IPv4: The IP forwarding is a fundamental operation performed by the router. The IPv4 uses the header of the packet to determine the destination address. A lookup is performed based on the destination address in

Table I. Time Taken by FEADS for Different Benchmarks

Application	Exec. Time (in Sec.)
IPv4	24.7
NAT	33.0
IPSec	30.9

the IP to determine the destination port number and the next hop address. The lookup table is stored in the SRAM. The time to live field in the IP header is decremented and the cyclic redundancy checksum (CRC) is recomputed. The packet is then forwarded to the next hop. The ADAG in this case consists of 23 nodes, five of which are memory nodes.

NAT: The NAT is a method by which many network addresses and their TCP/UDP ports are translated into a single network address and its TCP/UDP ports. A translation table stores the corresponding translation from the private IP address and port number to the globally visible router IP address and a unique port number. The translation table is stored in SRAM. The ADAG for NAT consists of 31 nodes, with seven memory nodes.

IPSec: The IPSec protocol is used to provide privacy and authentication services at the IP layer. IPSec supports two protocols depending on the level of security. A 32-bit connection identifier, referred to as a Security Parameter Index (SPI), contains a shared key used for encryption and the algorithm used for encryption. We assume that the SPI is stored in SRAM. The ADAG for IPSec consists of 29 nodes, six of which are memory nodes.

We consider the ADAGs for these applications (IPv4, IPSec and NAT) as input to the FEADS. These ADAGs are shown in Fig. 5. The sizes of the ADAGs themselves vary from seven to 11 nodes, excluding the communication nodes which may be introduced during mapping. Due to the small sizes of the ADAGs under consideration, converging to a solution using simulated annealing takes less than 30 s of runtime on a standard PC (Pentium 4 with 1 GB RAM operating at 3 GHz) even while considering multiple instances of the ADAG. The execution time of the FEADS framework for the different benchmarks are shown in Table I.

We do not consider other benchmark applications for the following two reasons. First, the ADAGS for other applications are not readily available and, at this point, we do not have a tool to generate the same automatically. Second, most of the network processing applications show very similar behavior, at least within the class of header processing and payload processing applications, in terms of their program flow, accessing SRAM, and DRAM memory, application specific units, etc. Hence we consider the ADAGS for IPv4, NAT, and IPSec as representative of commonly used network processing applications.

The IPv4 and NAT were scheduled on an IXP2400 processor, while IPSec was scheduled on IXP2850 since it uses crypto units for packet processing. The clock speed for IXP2400 is 600 MHz and that for IXP2850 is 1400 MHz. In all cases, we assume packet arrival is exponential distributed[3] with a mean arrival corresponding to a 5 Gbps line for IPv4 and NAT, and over a 10 Gbps line for IPSec. In all experiments, we use minimum size packets (64 bytes) as is customary in the evaluation of network processors(2) as this corresponds to the performance of the system under denial of service (DoS) attacks.(5)

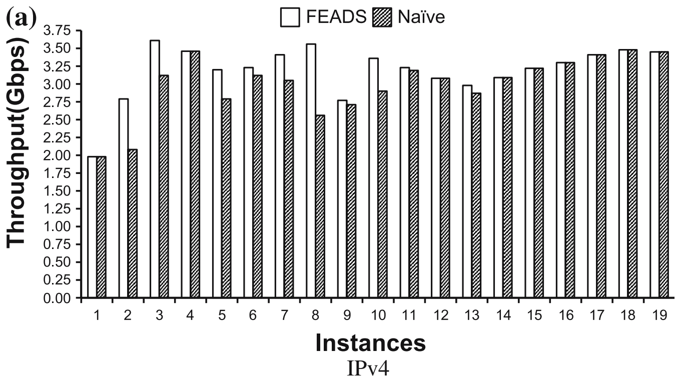
The results for various scenarios that we evaluated are presented in the following subsections.

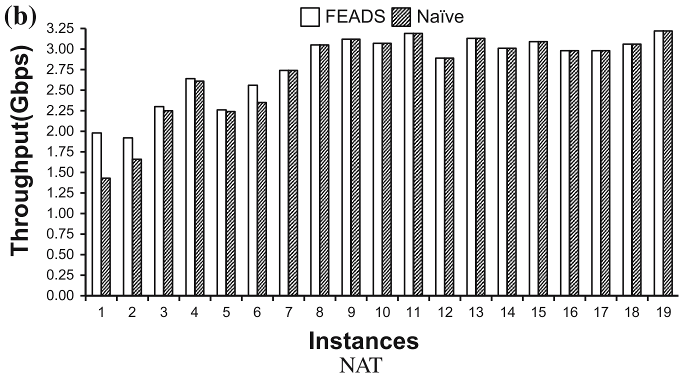
5.2. Naive versus FEADS Mapping

First, we compare FEADS with a naive binding scheme in which all tasks for a given instance of the application are bound to a single ME. Multiple instances are bound to different MEs in a round robin fashion. This results in an even distribution of tasks across microengines with no communication cost among the tasks. This is similar to the task assignment policy followed in Ref.(6). The tasks were scheduled using decomposed software pipelining. Note that although we call this scheme as *Naive*, only the binding is naive; the scheduling of tasks (within a ME) is still software pipelined. Thus a comparison of FEADS with the Naive scheme brings out the benefits of task binding performed by FEADS.

As the number of instances are increased, the throughput for both FEADS and naive mapping improves upto a certain extent (refer to Fig. 10). Beyond this the throughput fluctuates as the number of instances increases. This happens due to the ceil function described in Eq. 1. We see that the FEADS mapping achieves a higher throughput (7% in IPv4, 4% in NAT, and 9% in IPSec) compared to naive mapping. When fewer instances of the application are scheduled, FEADS performs much better than the naive mapping as the latter loads only a few microengines while the others are left idle. As the number of instances are increased, both naive and FEADS mapping perform comparably. This is because tasks are

[3]The performance of FEADS under bursty traffic has been evaluated and reported in Section 5.4.





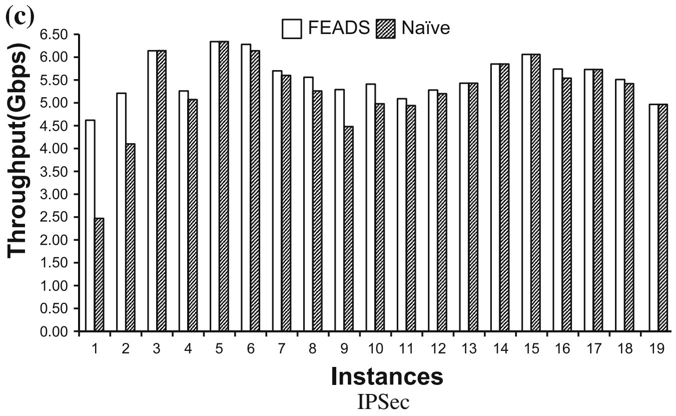


Fig. 10. Comparison of FEADS Mapping with Naive Mapping.

more evenly balanced across all resources and the schedule thus obtained is equal in length to that computed by FEADS.

5.3. Scheduled versus Immediate Execution

We have evaluated the mapping generated by the FEADS framework with and without imposing the static schedule generated by the cyclic scheduler on the PN model.

In the former case, arriving packets are held in the receive buffer until it is time to schedule the packet. Scheduling of packets happens at multiples of II time steps. In the case of immediate execution, packets are scheduled as soon as they arrive. The PN model generated by the scheduler model was simulated using the CNET(23) simulator. The throughput and buffer requirements (in RFIFO) in either case were obtained from the model. The results for the two cases is shown in Fig. 11, and the buffer requirements are compared in Fig. 12. Static scheduling of tasks results in 14% higher throughput on an average. Specifically, it achieves an improvement of 16% in IPv4, 13% in NAT, and 14% in IPSec over immediate scheduling. Note that the performance improvement is consistent across applications for all instances. Although the buffer requirement of FEADS schedule is higher than that of Immediate schedule, the maximum buffer requirement is still less than 64 packets in all cases. This corresponds to a buffer size of 4 KB, which is quite common. When we consider packets of size larger than 64 bytes at the same line rate, the interarrival time between packets is higher and hence the buffer requirements will be smaller.

5.4. Performance of FEADS under Bursty Traffic

Further, we evaluated the performance of FEADS for bursty traffic, wherein packets over the line arrive in bursts with periods of idle time. Specifically, the bursty traffic is is modeled as an aggregate of different substreams.(15) Each substream is assumed to be of constant packet size with finite ON/OFF periods. In the ON time each sub stream generates packets of constant size and specified line rate and restarts the packet generation after the OFF period. The arrival rate in the ON period within a substream is equal to $(packetsize)/(linerate)$. The ON and OFF times of each sub stream are Pareto distributed with a probability distribution function $f(x)$ given by

$$f(x) = \alpha\beta^{\alpha}/x^{\alpha+1}, \tag{2}$$

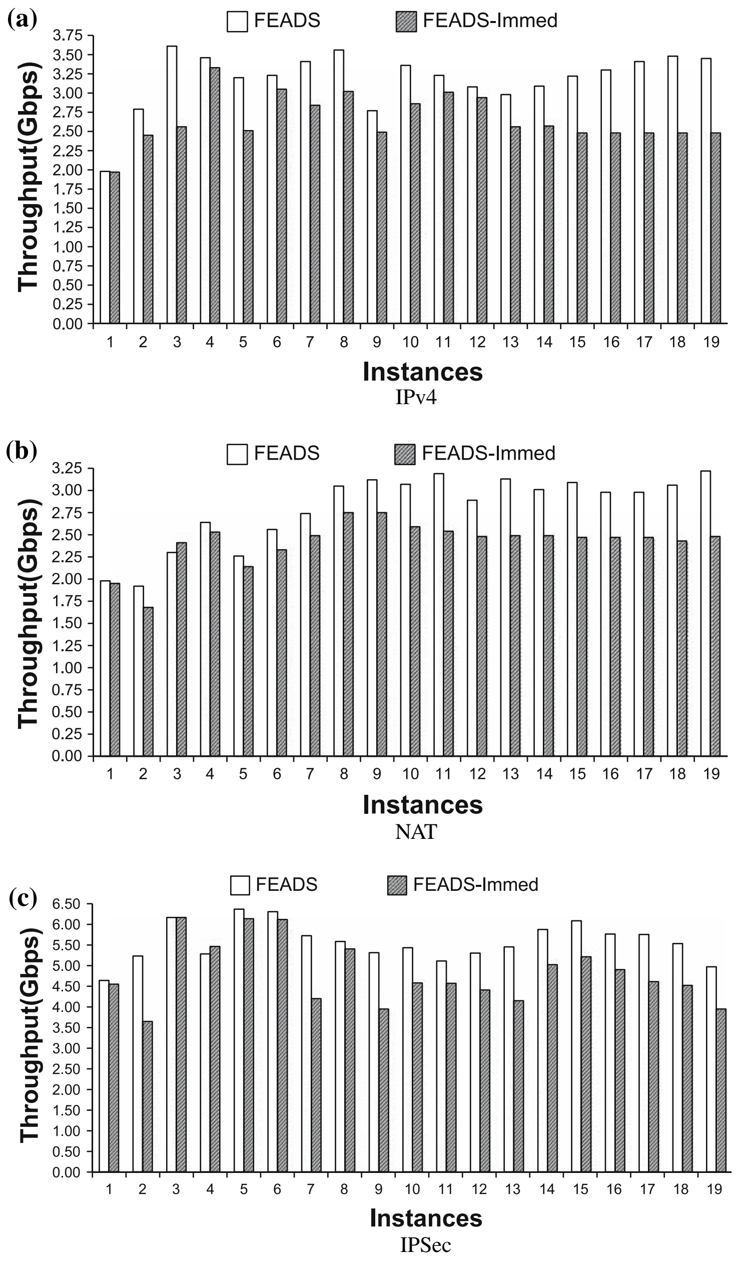


Fig. 11. Comparison of scheduled versus immediate execution.

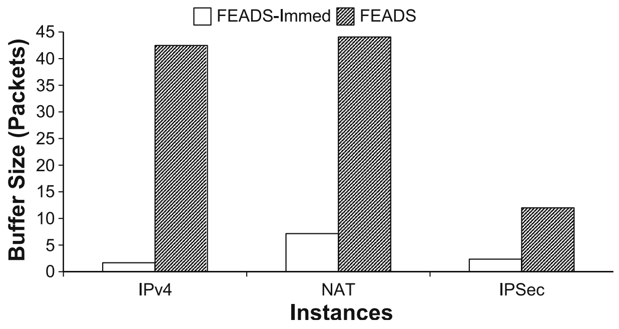


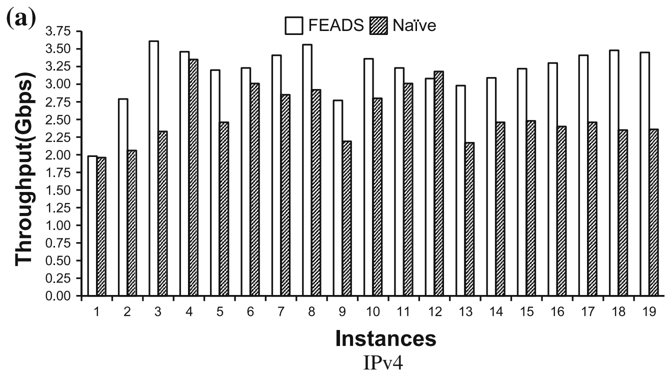
Fig. 12. Buffer requirements for scheduled versus immediate execution.

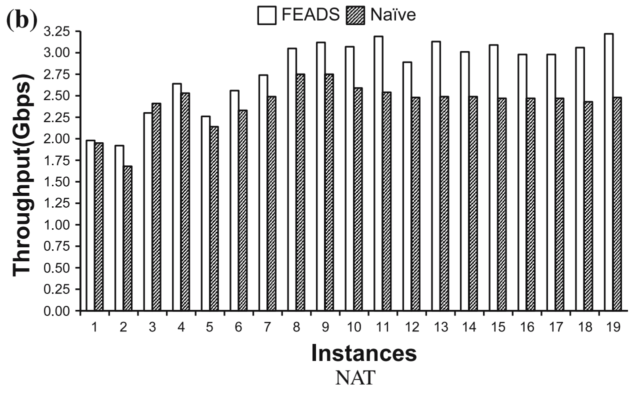
where α represents the shape parameter and β represents the scale parameter. The shape parameter α takes a value between 1 and 2. The scale parameter β is the ON/OFF time. The total line rate of the bursty traffic can be controlled by the line rates of the individual substreams and the the ON periods.

It has been shown that the traffic generated using the above methodology is bursty and similar to the traffic commonly encountered by routers.(15) The above traffic generator is modeled using PN. The resultant traffic generated is used as the input to the network processor. The performance results under the bursty traffic are shown in Fig. 13. We observe a similar trend in performance as in the earlier case. Specifically, FEADS achieves an improvement of 26% in IPv4, 21% in NAT, and 17% in IPSec on the average.

5.5. Imposing Code Size Constraints

Next we study the effect of code size constraints on the binding of tasks to resources. Imposing code size constraints limits mapping the entire application to a single microengine. Thus, in this case, a naive binding might not result in an even distribution of tasks. The throughput obtained by the naive binding scheme as compared to FEADS is shown in Fig. 14 for a code store constraint of 50 instruction words per ME. We observe that FEADS results in an increase in the throughput by 53% in the best case for IPSec and 12% on an average over all applications under consideration. The IPSec consists of a larger number of processing tasks than other applications and hence a naive binding does not exploit the available microengines and threads effectively.





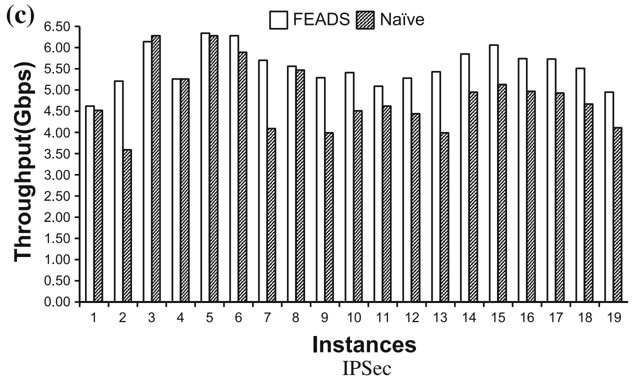


Fig. 13. Throughput with bursty traffic.

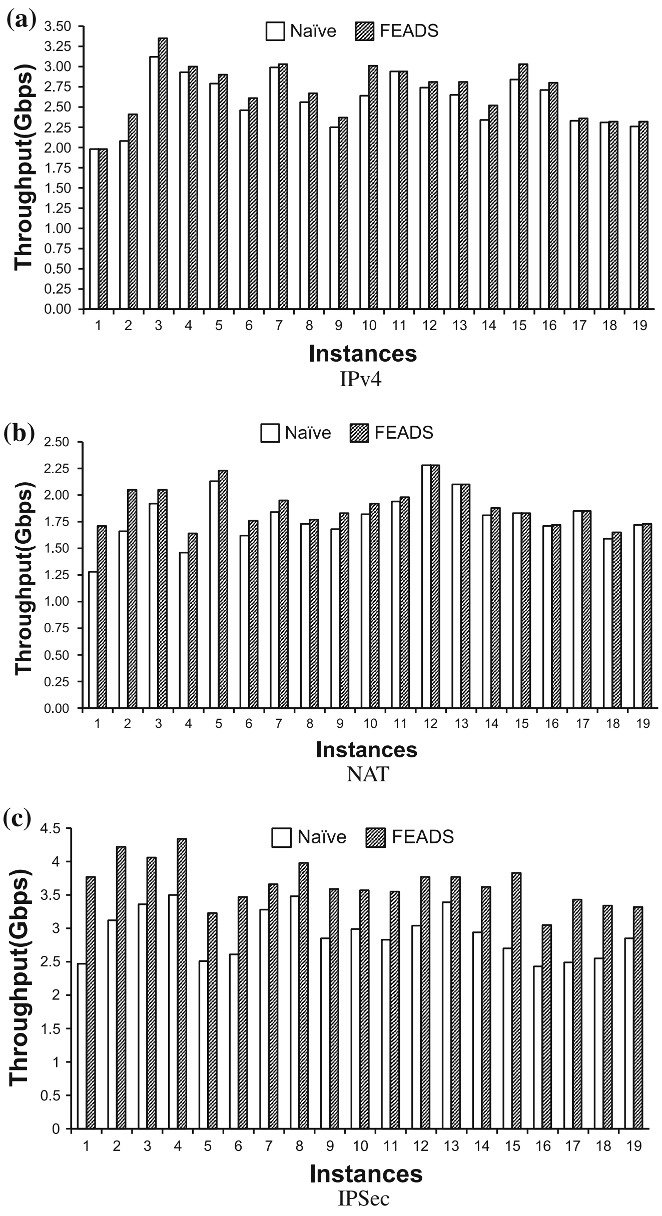


Fig. 14. Throughput with code size constraints.

5.6. Performance of Non-linear ADAGs

The ADAGs for the applications under consideration thus far are linear by nature. With such linear ADAGs, performing manual mapping is straightforward (with or without code constraints) and often results in the same mapping as in FEADS. We model more complex task graphs to better reflect the increasing complexity of network applications being implemented on network processors. We compared the performance of our scheme for two non-linear ADAGs, FEADS_input1 and FEADS_input2. The ADAG for FEADS_input1 was constructed using two parallel branches, one branch each from IPv4 and NAT. The throughput obtained by the naive scheme and from FEADS are shown in Fig. 15. In case of FEADS_input2, the throughput obtained by FEADS is upto 2.5 times higher than the naive scheme. Also we observe that the performance improvement is consistent for all number of instances. The results clearly indicate that with the increasing complexity of network applications, manual partitioning and binding of tasks is no longer sufficient to achieve the necessary performance.

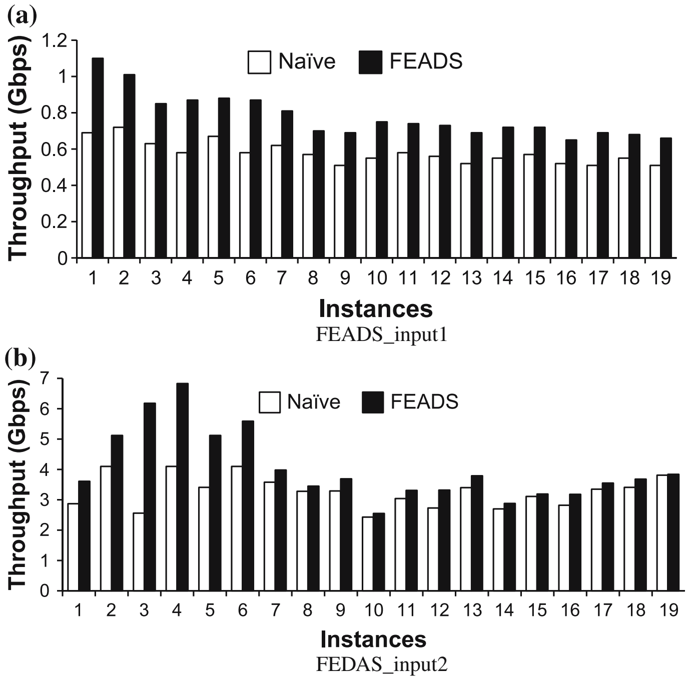


Fig. 15. Throughput for non-linear ADAGs.

5.7. Summary

From the above results, we observe that static task scheduling is beneficial and should be enforced when buffer requirements can be met. While manual binding of tasks to resources can result in a fairly good solution for linear graphs and without additional code-size constraints being imposed, it becomes increasingly difficult when either of these conditions is not met. FEADS yields a mapping whose throughput is comparable to that obtained by manual partitioning and scheduling for linear task flow graphs and upto 2.5 times higher for more complicated task graphs, and therefore effectively automates the process of application development on network processors.

6. Conclusions

This paper presents FEADS, a framework for automating the process of task partitioning, mapping and scheduling on network processors. The FEADS uses an ADAG with tasks of finer granularity than those proposed by previous approaches. It obtains an optimal mapping and scheduling of tasks onto resources using simulated annealing. Task scheduling is done using cyclic and r-periodic scheduling. We find that FEADS generates mappings whose throughput is comparable to that obtained by manual binding for linear ADAGs and upto 2.5 times higher for non-linear ADAGs. Further, imposing a static schedule on the generated mapping results in 14% higher throughput compared to unscheduled execution. The FEADS therefore provides an effective solution for mitigating the growing complexity of designing network applications to meet line speeds.

Automation of the ADAG generation module from a given application specification can be undertaken for future work. The framework can also be extended for mapping, scheduling and performance evaluation of applications for other network processor families.

7. ACKNOWLEDGMENTS

We acknowledge the Consortium for Embedded and Internetworking Technologies (CEINT) and Arizona State University, Tempe, USA for the partial support provided through a research subcontract. We are thankful to Govind Shenoy for his help in the PN model and the members of the High Performance Computing laboratory for their useful suggestions and comments. Lastly, we are grateful to the reviewers for their useful comments.

REFERENCES

1. T. Blickle, J. Teich, and L. Thiele, System-Level Synthesis Using Evolutionary Algorithms, in *Design Automation for Embedded Systems*, 1998.
2. M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-La, Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming, in *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, June 2005.
3. R. Ennals, R. Sharp, and A. Mycroft, Task Partitioning for Multi-Core network processors, in *International Conference on Compiler Construction (CC)*, 2005.
4. M. Franklin and S. Datar, Pipeline Task Scheduling on Network Processors, in *Proceedings of Third Workshop on Network Processors*, 2004.
5. L. Garber, Denial of Service Attacks Rip the Internet. *IEEE Comput*, **33**(4): 12–17 (2000).
6. S. Govind and R. Govindarajan, Performance Modeling and Architecture Exploration of Network Processors. in *Proceedings of the International Conference on Quantitative Evaluation of Systems*, Torino, Italy, 2005.
7. T. C. Hu, Parallel Sequencing and Assembly Line Problems, *Oper. Res.*, **9**(6): 841–848 (1961).
8. Intel Corp, Intel IXP1200 Network processor family.
9. Intel Corp, Intel IXP2400 Hardware reference manual.
10. Intel Corp, Intel IXP2800 Hardware reference manual.
11. Intel Corp, Intel IXP2850 Hardware reference manual.
12. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, Optimization by Simulated Annealing., in *Science* **220**, 598 (1983).
13. C. Kulkarni, M. Gries, C. Sauer, and K. Keutzer, Programming Challenges in Network Processor Deployment. in *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES03)*, 2003.
14. M. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, in *Conference on Programming Language Design and Implementation*, 1988.
15. W. E. Leland, M. S. Taqqu, W. Willinger and D. Wilson, On the Self Similar Nature of Ethernet Traffic., *Proc of SIGCOMM*, Sept 1993.
16. Lucent Technologies Inc, PayloadPlus Fast Pattern Processor, Apr. 2000. http://www.agere.com/support/non-nda/docs/FPPProductBrief.pdf
17. R. Morris, E. Kohler, J. Jannotti, and M.F. Kaashoek, The Click Modular Router, *ACM Trans. Comput. Syst.*, **18**(3): 263–297 (2000).
18. N. Shah, W. Plishker, and K. Keutzer. NP-Click, A Programming Model for the Intel IXP1200, in *Proceesings of Second Workshop on Network Processors (NP2) at the Ninth International Symposium on High Performance Computer Architecture (HPCA9)*, 2003.
19. L. Thiele, S. Chakraborty, M. Gries, and S. Kunzli, Design Space Exploration of Network Processor Architectures., in *Proceedings of First Workshop on Network Processors at the Eighth International Symposium on High Performance Compter Architecture (HPCA8)*, 2002.
20. V. Van Dongen, G. Gao, and Q. Ning, A Polynomial Time Method for Optimal Software Pipelining., in *Proceedings of the Second Joint International Conference on Vector and Parallel Processing: Parallel Processing*, 1992.
21. J. Wang and C. Eisenbeis, Decomposed Software Pipelining: A New Approach to Exploit Instruction Level Parallelism for Loop Programs., in *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Vol. A-23, 1993.

22. N. Weng and T. Wolf, Pipelining versus Multiprocessors: Choosing the Right Network Processor System Topology, in *Proceedings of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004) in conjunction with the 31st Annual International Symposium on Computer Architecture (ISCA 2004)*, June 2004.
23. W. M. Zuberek, Modeling Using Timed Petri nets — Event-Driven Simulation, Technical Report No. 9602, Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada, 1996. ftp://ftp.ca.mun.ca/pub/techreports/tr-9602.ps.Z.