Profiling and Mapping of Parallel Workloads on Network Processors

Ning Weng
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003 USA
nweng@ecs.umass.edu

Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003 USA
wolf@ecs.umass.edu

ABSTRACT

Network processors are embedded system-on-a-chip multiprocessors that are optimized to perform simple packet processing tasks at data rates of several Gigabits per second. To meet the performance demands of increasing link speeds and more complex network applications, network processors are implemented with several dozens of processor cores and execute multiple packet processing applications in parallel. The complexity of such systems makes it increasingly difficult for application developers to map applications to the various system resources and achieve optimal performance. We propose an automated profiling and mapping methodology for these highly parallel, embedded systems that starts out with a simple uniprocessor implementation of the networking application. An architecture independent representation of the runtime behavior of the application is used to map and schedule different processing steps to the underlying hardware. An analytic performance model is used in the process to estimate system performance and to find an near-optimal solution through iteration.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance—*Performance Analysis and Design Aids*; I.6.3 [**Computing Methodologies**]: Simulation and Modeling—*Applications*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*

Keywords

Embedded systems, network processors, application profiling, multiprocessor scheduling

1. INTRODUCTION

The Internet has progressed from a simple store-an-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 on routers. To provide the necessary flexibility and throughput for increasingly complex network services, routers are equipped with programmable "network processors" (NPs). These NPs are typically embedded systems-on-a-chip with dozens of parallel processor cores, several memories, and I/O components. Packet processing tasks are performed on the network processor before the packets are passed on through the router switching fabric and on to the next network link. This process is illustrated in Figure 1.

To meet the processing demands of increasing link speed and application complexity, network processors employ large number of parallel processing engines. These processors are typically arranged in a physical or logical topology that range from a simple pipeline to hybrid pools of pipelines. The real-time nature of processing network traffic requires that NP applications are finely tuned to achieve the required throughput performance. With increasingly complex hardware topologies that exhibit numerous non-obvious interactions between components, it becomes difficult for application developers to write, schedule, and optimize applications.

Current software development tools for network processors require a manual partitioning of applications as well as a manual assignment to processing engines. In our work, we propose a methodology to automatically profile network processor applications from a uniprocessor implementation and derive an architecture independent representation. This representation can then be mapped to the processing resources on a network processor. It is important to note that our methodology can be applied to a broad range of network processor topologies and is not limited to any particular commercial product.

The methodology that we employ is shown in Figure 2. The central components of the work are the profiling and scheduling steps. A configurable NP hardware description allows this process to be used on any current or future NP systems. The key technical challenges of this process that are addressed in this paper are the profiling and application mapping. In particular, we focus on issues of:

- **Abstract Application Representation.** A network processing application needs to be represented in a way that it can be easily 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.

- **Application Mapping.** Once we have the application represented as an ADAG, the next step is to map the ADAG onto

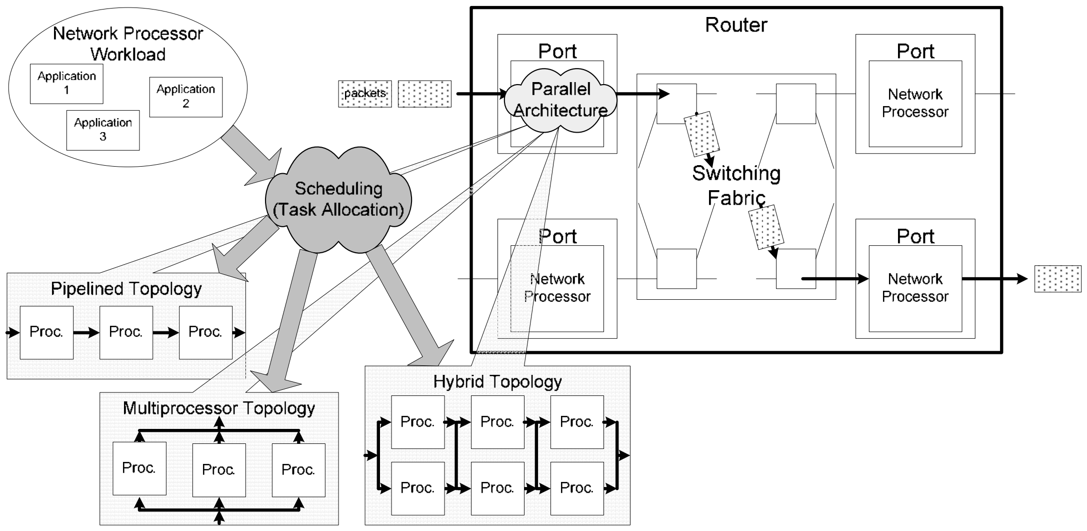


Figure 1: Workload Mapping on Network Processors. Network processors are implemented as system-on-a-chip multiprocessors. Applications need to be mapped to the architecture in an efficient way.

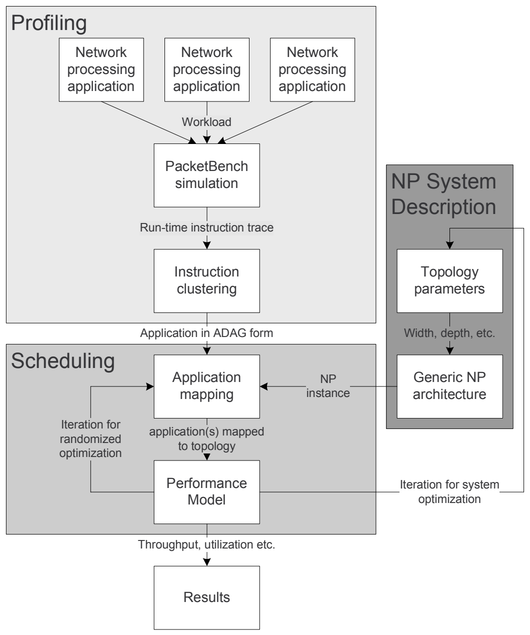


Figure 2: Methodology for Workload Mapping.

a NP topology. The goal is to optimize system performance while considering all the application dependencies. To approximate a solution to this NP-complete mapping problem, a randomization algorithm is used that achieves a good approximation. In order to evaluate the throughput performance of a given solution, an analytic performance model is used, which considers processing, inter-processor communication, memory contention, and pipeline synchronization effects.

The key differences of network processors as compared to conventional multiprocessing systems are the properties of the workload. In network processing, the workloads are simple (a few thousand instructions per packet) and highly regular (identical protocol processing for most packets). This allows us to use the proposed approach, which profiles and analyzes the workload at a level of detail that is not feasible for other parallel application (e.g., scientific or server applications). The regularity of processing also allows us to generate a task allocation schedule that can be adhered to in almost all cases. Packets that require special processing are deflected to the "slow path" of the router, where a special control processor handles the exception. In this paper we focus only on the fast path of the router.

Despite the simplicity of network processing, high network link rates present a significant performance challenge. Due to the embedded nature of network processors, many interactions on the NP can have an impact on the overall throughput performance (e.g., locking, access to shared data structures, access to shared co-processors). That is why a carefully crafted processing schedule can ensure that the system operates at the highest possible efficiency.

The remainder of the paper is organized as follows. Section 2 present related work. Section 3 presents our profiling approach. The results from the profiling are used in the scheduling presented in Section 4. Section 5 presents experimental results and Section 6 concludes this paper.

2. RELATED WORK

Several development environments and programming abstractions for network processors have been proposed. Most network processor vendors provide software development kits for their architecture that use modular programming abstractions. In the SDK for the Intel IXP2400 network processor [7], an NP used widely in academia and industry, a network processing application is built by combining several modules that handle ingress processing, forwarding, scheduling, and egress processing [5]. Similar modularity is exhibited by Teja [19], another SDK for the Intel IXP network processors. Other programming models for NPs include NP-Click proposed by Shah et al. [18], which is an extension to the Click modular router [9].

The major drawback of most of these approaches is that they require the application developer to have an in-depth understanding of the NP system architecture. This will become an increasingly pressing problem as network processors—as well as other embedded systems—move towards parallel architectures with large numbers of parallel processing resources. Our approach to solving this problem is a "bottom-up" approach that differs from the above work insofar that we use profiling information from a simple uniprocessor implementation of an NP application. This profiling information can be used to extract all available parallelism in the application, to map the application to processing resources, and to model the run-time performance through an analytic performance model. The bottom-up approach to analyzing and mapping workloads promises to significantly reduce the complexity with which the application developer has to deal.

Mapping and scheduling tasks on conventional multiprocessors has been explored by Austin and Sohi [1] and is conceptually similar to mapping tasks on an NP. However, there are significant differences in the underlying system architectures. Due to the embedded nature of NPs, there are practical limitations on how large the instruction store per processor can be (typically only a few thousand instructions with no memory hierarchy due to real-time constraints), how many interfaces are available for accessing off-chip memory (typically only few due to pin limitations of the packaging process), and what kind of inter-processor communication is possible (typically constraint by the system topology). Our approach to mapping tasks to NPs takes these constraints into account and utilizes several successful methods of multiprocessor mapping that were developed previously. Karp [8] and Motwani and Raghavan [12] showed that randomization in the context of mapping provides a good heuristic to solving the NP-complete mapping problem. More recently, Lakamraju et al. [10] have applied this idea to synthesizing networks that satisfy multiple requirements. Their result demonstrates that randomization is a powerful approach that can generate good results even with short run time.

We combine the randomized mapping approach with an analytic performance model for network processors to evaluate the performance of a given randomized solution. Franklin et al. have explored entire system-on-a-chip configurations for NPs through analytic performance modeling while considering area and power constraints [4]. Thiele et al. have proposed a performance model for network processors that considers the effects of network traffic [20] and Gries et al. have evaluated the performance/cost trade-offs for different network processor topologies based on a network calculus approach [6]. In this work, we use a performance model developed in our prior work that considers different processor topologies and considers several embedded system effects, like memory contention and pipeline synchronization [22].

3. WORKLOAD PROFILING

The goal of profiling the network processor workload is to generate an architecture independent application representation that can be used for network processor scheduling. We use a dynamic instruction trace to generate an annotated directed acyclic graph (ADAG), where nodes represent processing tasks and links represent control and data dependencies.

3.1 Instruction Trace

One key question is whether to use a static or a dynamic application analysis as the basis for this work. With a static analysis, detailed information about every potential processing path can be derived. All processing blocks can be analyzed—even the ones that are not or hardly used during run-time. A static analysis typically results in a "call-graph," which shows the static control dependencies (i.e., which function can call which other function).

A dynamic instruction trace analysis of the application shows exactly which instructions were executed and which instruction blocks were not used at all. In addition, all actual load and store addresses are available, which can be used for an accurate data dependency analysis. In this work, we use the dynamic instruction trace to reflect the run-time behavior of the application.

To obtain a trace of a realistic packet processing applications, we us a tool called "PacketBench" that we have developed to analyze network processing applications [15]. The profiling starts with the application written in uniprocessor C code, which is compiled with a cross compilation tool chain for the GNU C compiler and an ARM back-end. A PacketBench trace provides the registers and memory locations for all instructions that are executed as well as the control transfer information.

One subtle issue of this dynamic profiling approach is that each packet could potentially use a different execution path in the application. In some cases, certain blocks are executed a different number of times depending on the size of the packet (e.g., packet payload encryption). One solution is to assume that packets use the most common execution path and if the execution deviates from this case an exception is raised and processing is continued on the control processor. This is currently done on some network processors (e.g., IP option detection in an IP forwarding application). Our approach is to analyze a large number of packets and find the *union* of all execution paths. By scheduling the union on the network processor system it is guaranteed that all packets can be processed. The potential drawback is a lower system utilization as not all application components will be used at all times. However, our results show that network processing applications are very regular and simple. For most applications, over 90% of packets are covered by the most common execution path. When considering a union with 10–20% overhead over the common case, the execution of nearly all packets is possible [14]. This indicates that our approach of dynamic profiling and mapping of ADAGs is suitable for the network processor application domain.

3.2 ADAG Generation

Our bottom-up ADAG generation starts from a PacketBench trace that provides the processing and I/O requirements of each instruction block and the number of data dependencies between instruction blocks. Considering each individual data and control dependency, we obtain an initial DAG with thousands of nodes. To make the DAG tractable and useful for scheduling, we reduce the number of nodes by clustering multiple nodes to instruction clusters (i.e., processing tasks). We use a clustering technique called "ratio cut" introduced by Wei and Cheng [21]. Ratio cut has the nice property of identifying "natural" clusters within a graph without the need for a-priori knowledge of the final number of clusters.

The ratio cut, r_{ij}, for two clusters i and j is defined as the fraction of the sum of the data dependencies, d_{ij} and d_{ji}, and the product of the amount of processing performed in each cluster, p_i and p_j:

$$r_{ij} = \frac{d_{ij} + d_{ji}}{p_i \times p_j}. \quad (1)$$

This clustering method achieves a natural clustering of nodes through minimizing the inter-cluster dependencies, while maximizing cluster size of cohesive nodes.

The ratio cut algorithm is unfortunately NP-complete and thus not tractable for ADAGs with the number of blocks that we need to consider here. Therefore, we have developed a heuristic called "maximum local ratio cut" (MLRC) that is a greedy algorithm based on ratio cut and reduces the computational complexity while still achieving good results. The idea of MLRC is to find the pair of clusters that maximize r_{ij} (and thus show most cohesiveness) and join them.

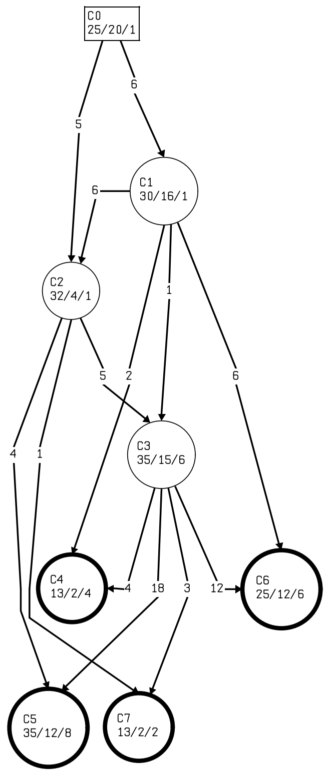


Figure 3: Annotated Directed Acyclic Graph (ADAG) for Flow Classification Application.

3.3 Example Workload

To illustrate the ADAG generation process as well as the mapping results below, we consider an example workload consisting of four typical network processors applications:

- **IPv4-radix.** IPv4-radix is an application that performs RFC1812-compliant packet forwarding [2] 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 [13].
- **Flow Classification.** Flow classification separates traffic 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 [3], which is the new Advanced Encryption Standard (AES).

One example ADAG that is obtained from the MLRC clustering of the Flow Classification is shown in Figure 3. The nodes contain a 3-tuple annotation with the number of instructions that are executed in this node, the number of reads to memory and the number of writes to memory. Data that is moved between processing tasks is shown as edge weights. Rectangular nodes identify the starting node, bold nodes indicate the terminating nodes.

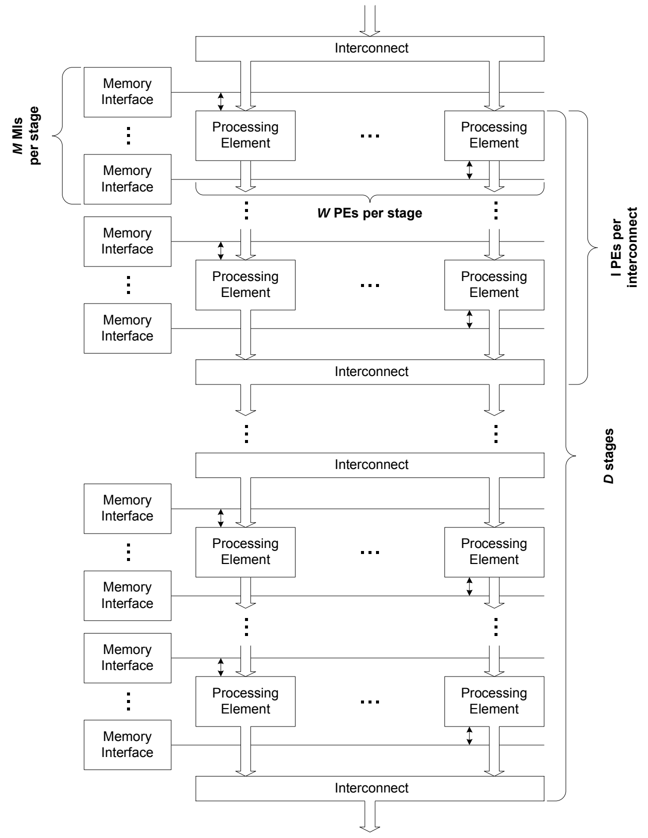


Figure 4: Generic Network Processor Architecture. The parameters which can be varied are shown in the figure.

4. WORKLOAD MAPPING AND SCHEDULING

With applications represented by ADAGs we are ready to map the workload to the network processor. To find a general solution for the mapping problem, we do not focus on a single commercial NP architecture, but a generic system that can be parameterized.

4.1 Parameterized Network Processor Architecture

Our general, parameterized network processor topology shown in Figure 4 consists of three components: processing elements, 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) and the number of memory channels shared by one row of processing elements (M).

These parameters enable us to represent a wide range of possible NP architectures. A parallel multiprocessor topology can be modeled by fixing the pipeline depth (D) and interconnect stages (I) to 1, while varying the pipeline width (W). A pipeline architecture can be modeled 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.

4.2 Randomized Mapping

The goal of the mapping is to assign processing tasks (i.e., ADAG nodes) to processing elements and generate a schedule that achieves the maximum system throughput. This assignment is not easy because the mapping process needs to consider the dependencies within an ADAG and ensure that a correct processing of packets is possible. Further, Malloy et al. have shown that producing a optimal schedule for a system that includes both execution and communication cost is NP-complete, even if there are only two processing elements [11]. Therefore 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 [8, 12, 10].

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 a 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.

4.3 Performance Model

In order to evaluate the performance of a given mapping and scheduling, an analytic performance model is used. The annotations of the ADAG give information on how many instructions are processed and how many memory accesses are performed by each processor. For shared resources (e.g., memory) it is important to consider the queuing effect from multiple parallel requests that can only be served in sequence. We have developed a model in prior work [22] with which the processing time of each stage of the NP pipeline can thus be determined.

The use of an *analytic* model for the evaluation of the randomized schedule is crucial. Randomized mapping performs best when a large number of solutions are tried and only a fast evaluation process allows this to happen. The use of more accurate by orders of magnitude slower simulation would not be suitable.

The resulting, near-optimal schedule can then be implemented on the network processors. Packets that are processed by the system are passed to the start node of the appropriate ADAG and processed in parallel or pipeline fashion (depending on the allocation and the architecture).

5. RESULTS

We present several results that show the performance of the task allocation mechanism discussed in Section 4 using the workload profiles from Section 3. First, we compare the mapping results with the theoretical upper bound to illustrate the effectiveness of our approach. We also present results showing the impact of instruction store limitations on system performance. Finally, we evaluate different NP architectures for their suitability for different applications.

5.1 Comparison to Ideal Schedule

The ideal schedule for an NP architecture consisting of W parallel processors (i.e., pipeline depth of 1) depends on the number of instructions, $instr_A$, and memory accesses, r_A and w_A, that ADAG A requires. The average memory access delay (queuing and transmission) is given by mem_{delay} and determined through a Machine Repairman Model [17] [16]. Thus, the ideal throughput is:

$$throughput_{ideal} = \frac{clk}{\frac{instr_A}{W} + mem_{delay} \times (r_A + w_A)}. \quad (2)$$

Table 1 show the ideal throughput as compared to actual scheduling results for the four workload application ($W = 8$ and $D = 1$). The actual throughput that is achieved by our mapping algorithm is around 1% less. Since only complete ADAG nodes can be allocated to processing elements, the applications cannot be distributed entirely evenly over all processing elements and some processing stalls are introduced. Nevertheless, a near-optimal schedule is achieved by randomized mapping.

5.2 Limitations on Instruction Store

One main limitation on current commercial network processors is the amount of instruction store that is available to each processing element (typically only a few thousand instructions). While the *static* instruction store does not directly translate into the *dynamic* instructions shown in Table 1, there is still a limit on how many ADAGs can be mapped to current NPs.

Figure 5 shows the throughput performance of two applications for different number of parallel ADAGs (x-axis) for different architectures. The key observation is that for a small number of ADAGs (i.e., limited instruction store) the pipelined architectures perform better. For larger numbers of ADAGs (i.e., larger instruction store) the parallel architectures perform better. This is again due to the need for even distribution of processing tasks to achieve optimal performance. On a parallel architecture without pipelining it is not possible to fully utilize all processing elements when the number of ADAGs that can be scheduled is limited to a small number. If the number of ADAGs exceeds W, good throughput can be achieved because all processing elements can be utilized. Pipelined architectures perform better for small numbers of scheduled ADAGs. With an increasing number of ADAGs communication between processing elements increases and limits the throughput.

5.3 Network Processor Topology Exploration

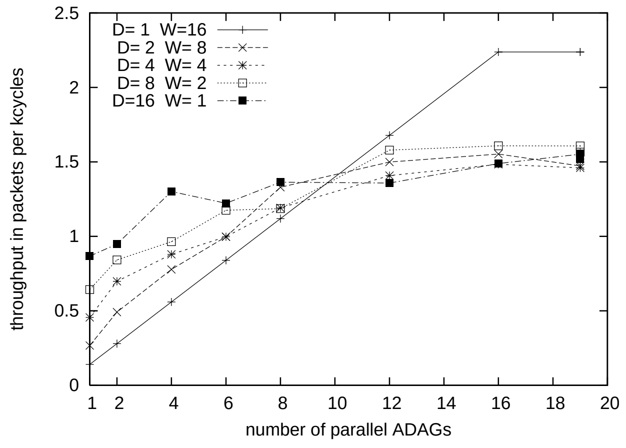
One of the key architectural aspects of a network processor is the system topology that determines how processing engines are interconnected and how parallelism is exploited. As shown above, the choice of topology does have a considerable impact on the overall performance. We can use the scheduling and performance model methodology for design space exploration to understand basic tradeoffs between different topology.

Figure 6 shows the throughput performance of NP topologies with configurations of $D = 1 \ldots 16$ and $W = 1 \ldots 16$. The memory channels are set to $M = 2$ per stage. As Figure 5 shows, the performance increases with the number of ADAGs. To put a limit on the NP architecture, the maximum number of ADAGs is limited to 400 and the maximum number of processing instructions is set to 10,000 per processing element.

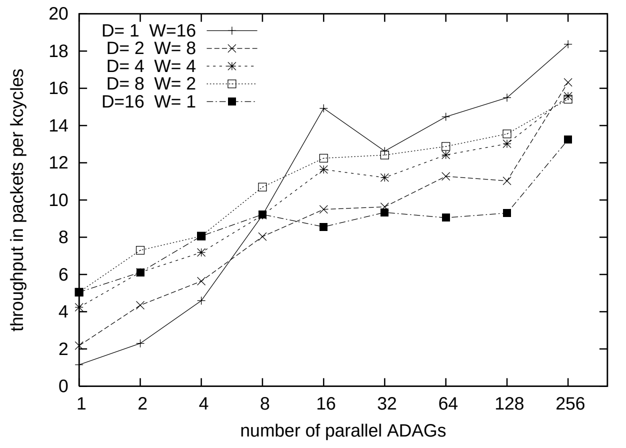
For the pipeline width, there is limited growth with increasing

Table 1: Ideal vs. Actual Performance for Different Applications.

Application	Instructions	Memory accesses	Ideal throughput *in pkts/kcycl*	Actual throughput *in pkts/kcycl*	Efficiency *in %*
IPv4-radix	4228	292	0.342	0.340	99.42%
IPv4-trie	201	67	1.493	1.492	99.94%
Flow Class.	208	112	0.892	0.883	98.99%
IPsec	2662	454	0.220	0.218	99.09%

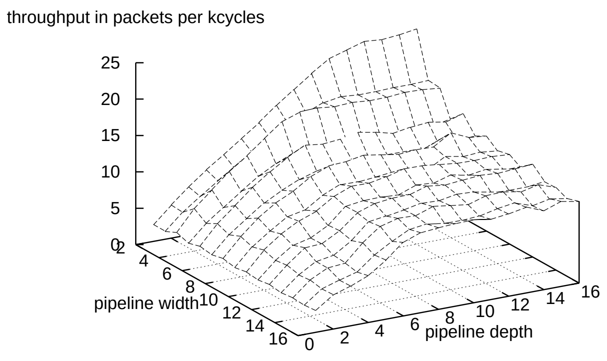


(a) IPv4-radix

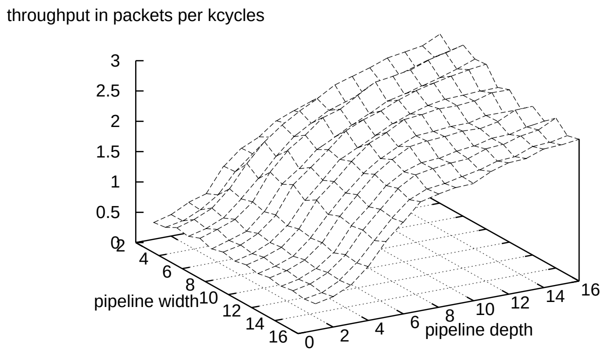


(b) IPv4-trie

Figure 5: Number of ADAGs Depending on Topology. The memory service time is $S = 10$.



(a) IPv4-trie



(b) IPsec

Figure 6: Throughput Depending on Topology. The number of memory interfaces per stage is $M = 2$ and the memory service time is $S = 10$.

numbers of processors. This is due to contention on the two memory channels per stage, where the off-chip memory access time dominates the overall stage time. For all applications (Flow Classification and IPv4-radix not shown), the performance increases as the pipeline depth increases and levels off at larger number of pipeline stages. This is due to the limit on the number of ADAGs and instructions per processing element. Otherwise the growth would continue up to the point where communication becomes a bottleneck.

These results lead to two conclusions regarding NP topologies: (1) memory accesses dominate as performance bottlenecks and (2) pipelining requires large instruction stores to allow for a balanced scheduling. The observation that performance improves with increasing pipeline depth and that memory interfaces saturate with increasing pipeline width are not surprising per se. Nevertheless, they serve as sanity check for the correct behavior of our framework and the effectiveness of randomized mapping.

The presented results show the versatility of our framework to obtain an understanding of performance tradeoffs between different application mappings and different network processor topologies.

6. CONCLUSION

In this work, we have introduced a methodology for profiling and scheduling networking workloads on highly parallel network processor architectures. The scheduling is based on randomized mapping, which is a good heuristic to solve the NP-complete scheduling problem. The results show that this approach achieves near-optimal schedules and works for a broad range of network processor architecture. The results can further be used in network processor design to optimize NP system topologies.

The limitations of this work are in some of the assumptions made in the performance model. The randomized mapping approach requires a fast evaluation of different configurations, which necessarily requires an analytic performance model rather than more detailed simulations. To keep the analytic performance model tractable, a number of simplifications had to be made (e.g., first come first serve memory access). It is important to note, however, that more detailed models can be used – which we plan to develop in future work – and thus more accuracy be achieved. The fundamental methodology of dynamic profiling and ADAG mapping is not effected.

We believe that the methodology that we have presented poses a promising approach to managing the complexities of highly parallel network processors and embedded systems in general. The profiling and scheduling can be done entirely automatically from a uniprocessor implementation and dynamic instruction trace of an application. It is conceivable that such a functionality will become part of software development kits and operating systems of future network processors and other parallel embedded systems.

Acknowledgements

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

7. REFERENCES

[1] T. M. Austin and G. S. Sohi. Tetra: evaluation of serial program performance on fine-grain parallel processors. Technical Report 1163, Computer Science Department, University of Wisconsin, Madison, WI, July 1993.

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

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

[4] 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, pages 29–50. Morgan Kaufmann Publishers, Nov. 2003.

[5] S. D. Goglin, D. Hooper, A. Kumar, and R. Yavatkar. Advanced software framework, tools, and languages for the IXP family. *Intel Technology Journal*, 7(4):64–76, Nov. 2004.

[6] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer. Exploring trade-offs in performance and programmability of processing element topologies for network processors. In *Proc. of Second Network Processor Workshop (NP-2) in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 75–87, Anaheim, CA, Feb. 2003.

[7] Intel Corp. *Intel Second Generation Network Processor*, 2002. http://www.intel.com/design/network/products/np-family/ixp2400.htm.

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

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

[10] V. Lakamraju, I. Koren, 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.

[11] 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.

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

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

[14] R. Ramaswamy, N. Weng, and T. Wolf. Analysis of network processing workloads. Under submission.

[15] 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.

[16] 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.

[17] A. L. Scherr. An analysis of time-shared computer systems. Technical Report TR-18, Massachusetts Institute of Technology, Cambridge, MA, USA, 1965.

[18] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A programming model for the intel IXP1200. In *Proc. of Second Network Processor Workshop (NP-2) in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 100–111, Anaheim, CA, Feb. 2003.

[19] Teja Technologies. *TejaNP Datasheet*, 2003. http://www.teja.com.

[20] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In *Proc. of First Network Processor Workshop (NP-1) in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, pages 30–41, Cambridge, MA, Feb. 2002.

[21] 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.

[22] N. Weng and T. Wolf. Pipelining vs. multiprocessors - choosing the right network processor system topology. In *Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004) in conjunction with The 31st Annual International Symposium on Computer Architecture (ISCA 2004)*, Munich, Germany, June 2004.