

Analytic Modeling of Network Processors for Parallel Workload Mapping

NING WENG

Southern Illinois University Carbondale[†]

TILMAN WOLF

University of Massachusetts Amherst

Network processors are heterogeneous system-on-a-chip multiprocessors that are optimized to perform packet forwarding and processing tasks at Gigabit data rates. To meet the performance demands of increasing link speeds and complex network applications, network processors are implemented with several dozen embedded processor cores and hardware accelerators that run multiple packet processing applications in parallel. The parallel nature of the processing system makes it increasingly difficult for application developers to understand and manage resources and map processing tasks to the hardware. To address this problem, we present a methodology for profiling and analyzing network processor applications, mapping processing tasks to a generalized network processor architecture, and analytically determining the expected throughput performance. The key novelty of this work is not only the adaptation of application analysis and mapping algorithms to heterogeneous network processors but also that the entire process can be automated and hidden from the application developer. Starting with the analysis of a uniprocessor implementation of the application, the process yields a mapping of the partitioned application that shows best performance for a given network processor system. The simplicity of the proposed randomized mapping algorithm allows the use of this methodology in network processor run-time systems where dynamic reallocation of tasks is necessary but processing power is limited. We present results that show the effectiveness of the analysis and mapping methodology as well as its application to design space exploration.

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*

General Terms: Design, Performance

Additional Key Words and Phrases: Application profiling, embedded systems, multiprocessor scheduling, network processors

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 on routers.

[†]This work was performed while at the University of Massachusetts Amherst.

Authors addresses: Ning Weng, Department of Electrical and Computer Engineering, Southern Illinois University Carbondale, Carbondale, IL 62901; email: nweng@siu.edu; Tilman Wolf, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003; email: wolf@ecs.umass.edu.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 1539-9087/20YY/0000-0001 \$5.00

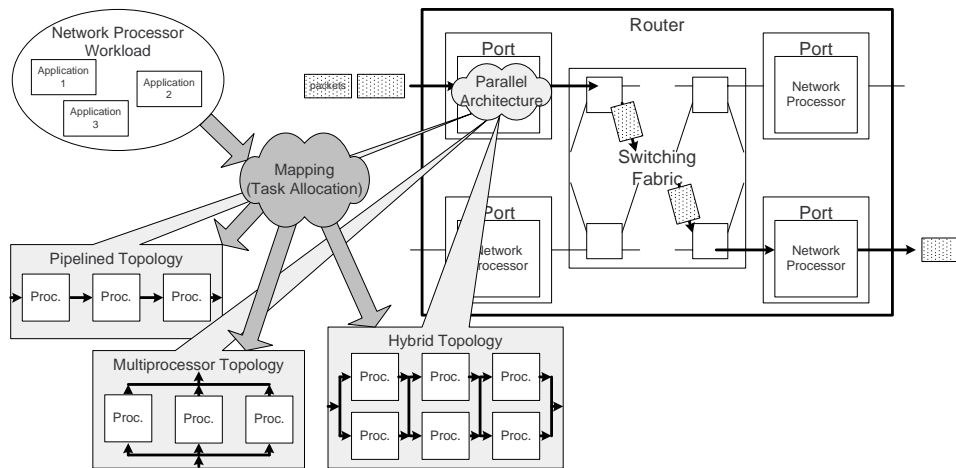


Fig. 1. Workload Mapping on Network Processors. Network processors are implemented as system-on-a-chip multiprocessors with a variety of system topologies. Applications need to be mapped to the architecture in an efficient way.

To provide the necessary flexibility for increasingly complex network services, routers are equipped with programmable “network processors” (NPs). These NPs are typically implemented as systems-on-a-chip with a number of processor cores, 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 through the next network link. This is illustrated in Figure 1.

The workload for network processors is unique and different from conventional workstation processors. Typical applications include forwarding of packets, packet classification, packet scheduling, and security application (e.g., firewall, intrusion detection). The workload is characterized by a large number of simple tasks and massive amounts of I/O operations. Processing a packet can range from simple IPv4 forwarding (i.e., route lookup and scheduling) to processing intense payload modifications (e.g., encryption or compression). While none of this processing is particularly complex, the Gigabit data rates that need to be supported by NPs generate significant performance demands.

The necessary performance of NPs is achieved through exploiting the inherent parallelism in network processing. Packets that belong to different network connections can be processed independently as a TCP/IP network makes no guarantees on packet order. Packets belonging to the same connection usually should be processed in-order for performance reasons, but this is not mandatory. As a result, almost arbitrary levels of parallelism can be achieved by replicating packet processing functionality on multiple processing engines and handling packets in parallel. Limits on this parallelism are imposed by the number of processing engines that can be implemented on a chip with given size, power consumption constraints, and contention on shared resources.

Current network processors are implemented with multiple processing cores, and parallelism is reflected in the architecture through the use of pipeline or multiprocessor configurations (see Figure 1). Typical network processors use eight to sixteen processor cores and potentially a number of special-purpose coprocessors. As link speeds and application com-

plexity increase and CMOS feature size decreases, it can be expected that next-generation network processors contain several dozen processing engines. The architectural aspects of such network processors are an important research area. An equally important question is how to program and operate such highly parallel processors while maximizing the available system performance. The focus of our work is to answer these questions.

With current software development tools and methodologies, it is challenging to program an application for an NP. The multiprocessor nature of an NP requires that the application developer partition the application and allocate it to processing resources by hand. To achieve better load balancing for higher throughput, this typically requires that parts of the application are programmed in assembly and fine-tuned for performance. 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 used to map workloads to the numerous processing resources on a network processor. Since an efficient mapping necessarily needs to consider system performance, we use an analytic performance model to evaluate each potential mapping. The three main contributions of this work are:

- (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.** With 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 application dependencies. To approximate a solution to this NP-complete problem, a randomized algorithm is used that achieves a good approximation.
- (3) **NP System Performance Modeling.** The system performance model is a key component for the randomized mapping algorithm. The model considers processing, inter-processor communication, contention on memory interfaces, and pipeline synchronization effects to determine the overall throughput.

Figure 2 shows our methodology for profiling and mapping network applications. The central components of the methodology are the ADAG generation, the application mapping onto the NP architecture, and the performance model. While this methodology looks like a generic approach to application mapping on parallel processor systems, the network processing environment poses important differences to existing work, which will be further discussed in Section 2:

—**Partition and Mapping Granularity.** The problem of partitioning applications appears in all parallel processor systems. The system details determine the granularity at which the partitioning is performed. High-performance processors and mesh-based multiprocessors exploit instruction-level parallelism (ILP). Grid-based computing systems partition application at much coarser levels and distributed complex functions to each processor. For network processing systems, we consider a granularity of a tens to hundreds of RISC instructions, which is limited by the instruction store of a typical network processor.

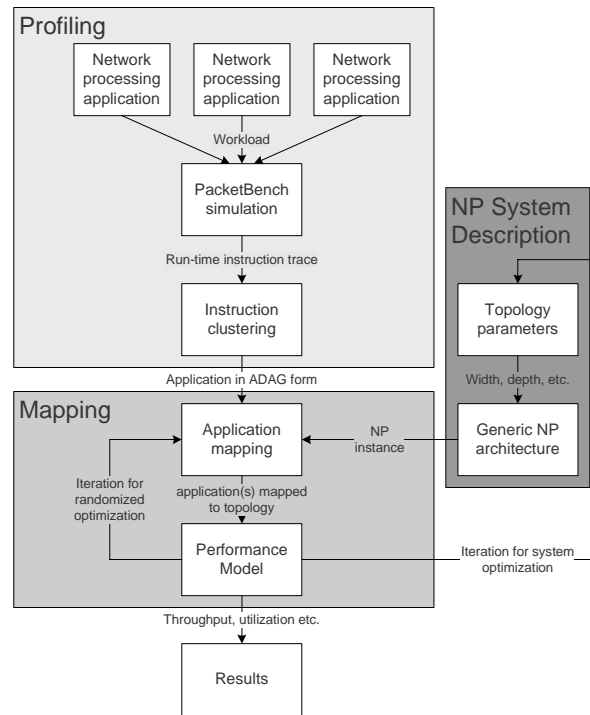


Fig. 2. Methodology for Workload Profiling and Mapping.

- Heterogeneity of System.** Practical network processor architectures are often equipped with co-processors and other mechanisms for application domain specific accelerators. The regularity of network processing ensures sufficient utilization of such components. For example, every IPv4 packet requires checksum computation, which is a significant portion of the overall processing requirements for IPv4 forwarding. This aspect requires an application domain specific analysis and mapping methodology.
- Heterogeneity of Workload and Dynamic Adaptation.** The network traffic processed by NPs presents a continually changing workload. While the issue of run-time management of network processors is not within the scope of this paper, it is important to consider the need for mapping multiple different applications on a single system (i.e., workload heterogeneity) as well as the need for dynamic run-time reconfiguration [Kokku et al. 2003; Wolf et al. 2005]. This means that application mapping and re-mapping needs to be performed during system operation (typically by the control processor of the NP system). Thus, the mapping process of any useful methodology needs to be of comparably low computation cost.

These three aspects are particular to the network processing domain and are considered in our analysis and mapping methodology.

With an automated way of profiling and mapping workloads onto any network processor system, we can also explore the more fundamental question of what NP system architectures should look like. Since there are a number of choices in the design of NPs (number

and types of processors and hardware accelerators, number of memory interfaces, type of interconnect topology), this is still an open question. Figure 2 illustrates our approach to addressing this problem, which involves modifying the topology parameters and comparing the throughput performance of different architectures.

The main contribution of this work is the methodology for application analysis, randomized mapping, and performance modeling for heterogeneous network processor topologies. We present a number of evaluations that show the effectiveness of this approach as well as related results that show how it can be used in design space exploration. In particular, we present the following results:

- Evaluation of speed, correctness, and quality of mapping algorithm.
- Performance tradeoffs between different network processor topologies that are due to system parameters (e.g., multithreading) and operational effect (e.g., communication and synchronization delay).
- Impact of instruction store limitations in realistic network processor systems on mapping and throughput performance.

The remainder of this paper is organized as follows. We discuss work related to NP architectures, application profiling, mapping, and performance modeling in Section 2. Section 3 introduces the application profiling and ADAG abstraction of network applications. Section 4 describe the randomized mapping algorithm. The analytic performance model for evaluating NP systems is presented in 5. Results are shown and discussed in Section 6. Section 7 summarizes and concludes the paper.

2. RELATED WORK

Network processors have been implemented as highly parallel multiprocessing systems in a variety of ways, ranging from a multiprocessor to pipelined and hybrid topologies. Commercial examples of network processors are the Intel IXP2400, the AMCC np7510, the EZchip NP-1, and the Agere Fast Pattern Processor and Routing Switch Processor.

To allow the programming of these NP multiprocessors, various development environments and programming abstractions have been proposed. Most NP vendors provide software development kits for their architecture that use modular programming abstractions. In the Intel SDK, for example, a network processing application is built by combining several modules that handle ingress processing, forwarding, scheduling, and egress processing. Current work in the area focuses on applying higher-level programming abstractions to simplify code development (e.g., domain-specific programming language by Teja [Teja Technologies 2003], C compiler for IXP by Intel [Goglin et al. 2003]). Other programming models for NPs include NP-Click proposed by Shah et al. [Shah et al. 2003], which is an extension to the Click modular router [Kohler et al. 2000].

The major drawback of most of these approaches is that they still require the application developer to have an in-depth understanding of the NP system architecture in order to tune the code for maximum performance. This will become an increasingly more pressing problem as network processors—as well as other embedded systems—move towards parallel architectures with dozens of parallel processing resources. In such systems, the allocation of processing tasks to resources needs to be done in an automated fashion without requiring the developer to make such decisions. Existing intermediate representations of network processor applications require programmers to manually partition applications

(e.g., communication-oriented data flow representation in NP-Click) or do not consider run-time information (e.g., call graphs). We address this problem with 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 is represented as a annotated directed acyclic graph (ADAG), which 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. This bottom-up approach of analyzing and mapping workloads promises to significantly reduce the complexity with which the application developer has to deal.

Partitioning and mapping of applications has been studied in the context of a variety of parallel processing systems. Fine-grained partitioning to exploit instruction-level parallelism has been shown in tiled or mesh-based architectures (e.g., the Raw microprocessor [Taylor et al. 2002]). Stream-based architectures partition the applications into very small “kernel” functions (e.g., the Imagine processor [Kapasi et al. 2003]). Grid computers partition large applications into tasks that are sufficiently complex to amortize the communication cost between autonomous computing systems [Foster and Kesselman 2004]. As discussed above, network processing poses several additional domain-specific constraint (granularity, heterogeneity, and dynamic adaptation) that are not considered in any of the above approaches.

Mapping algorithms for assigning task graphs to multiprocessors has been surveyed by Kwok and Ahmad [Kwok and Ahmad 1999]. Mapping tasks to conventional multiprocessors has been done by Austin and Sohi [Austin and Sohi 1993], is conceptually similar to mapping tasks inside an NP, but there are significant differences in the underlying system architectures. Similarly, scheduling in the high-performance computing domain [Dowdy et al. 1999] assumes abstract concepts of processor load and I/O resources that do not directly translate to network processors. 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 caching 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 while utilizing some successful methods for multiprocessor mapping developed previously. Karp [Karp 1991] and Motwani and Raghavan [Motwani and Raghavan 1995] showed that randomization in the context of mapping provides a good heuristic to solving the NP-complete mapping problem. More recently, Lakamraju et al. [Lakamraju et al. 2002] 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 times.

Using a heuristic approach with randomization to solve the mapping problem requires an efficient method for evaluating the performance of a given solution. Performance models for network processors have been developed in our own work [Franklin and Wolf 2002; 2003] as well as by Thiele et al. [Thiele et al. 2002] and Gries et al. [Gries et al. 2003]. One of the main components of such an embedded system model is the model for the delay caused by the contention on shared resources, like off-chip memory. Specific solutions for queuing delay and service times are provided in [Bhandarkar 1975; Hoogendoorn 1977; Grasso et al. 1984]. Methods have also been developed that provide lower bounds on

response times for shared resources with deterministic service time [van Gemund 1993].

Each of the three components of our methodology—profiling, mapping, and performance modeling—has been explored to some extent (and often in a different context) in prior work. The novel aspect of our contribution is the adaptation to the unique constraints of heterogeneous network processors (see Section 1) and the integration to an automated profiling and mapping process.

3. WORKLOAD PROFILING

The goal of profiling network processor workloads is to generate an architecture independent application representation that can be used for network processor mapping and 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 Static vs. Dynamic Application Analysis

Network processing applications are inherently very simple and execute a relatively small number of instructions (as compared to workstation applications) [Wolf and Franklin 2000]. This allows us to use much more detailed analysis methods that would be infeasible for large programs.

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 run-time analysis of the application (e.g., an instruction trace) 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. For mapping and performance modeling, run-time behavior yields more relevant information and thus is used in this work.

The drawback of run-time analysis is that each packet could potentially cause a different sequence of execution. To address the issue of variations in network traffic processing even within the same application (e.g., different packet services or number of loop executions), we analyze a large number of packets and find the union of all execution paths. By mapping the union on the network processor system it is guaranteed that all packets can be processed, but the drawback is a lower system utilization as not all components will be used at all times.

For the majority of applications that we have analyzed for this paper, the union of all execution paths is only a few percent larger than the majority of individual execution paths [Ramaswamy et al. 2005]. The few cases where a small fraction of packets require significantly more and different processing from the common case are not further considered. In a practical router system, these packets would be handled in the “slow path” by the control processor (e.g., for IP options or error handling).

3.2 Trace Generation

To obtain the run-time traces, we use our “PacketBench” tool that we developed specifically for analyzing packet processing applications [Ramaswamy and Wolf 2003]. A Pack-

Inst.-#	Address	Instruction	Effective Address
...			
129	33557096	: ldrb r3,[r4,#8]	: 0x33977912
130	33557100	: cmp r3,#0	: 0x-----
131	33557104	: bne 0x2000aa4	: 0x-----
132	33557156	: sub r3,r3,#1	: 0x-----
133	33557160	: strb r3,[r4,#8]	: 0x33977912
134	33557164	: mov r2,#65280	: 0x-----
135	33557168	: ldr r3,[r4,#8]	: 0x33977912
136	33557172	: add r2,r2,#254	: 0x-----
137	33557176	: mov r3,r3, lsr #16	: 0x-----
138	33557180	: cmp r3,r2	: 0x-----
139	33557184	: mov r2,#1	: 0x-----
...			

Fig. 3. Profiling Trace with Data and Control Dependencies.

etBench trace provides the registers and memory locations for all instructions that are executed as well as control transfer information.

A dependency analysis of the annotated run-time trace yields data and control dependencies between registers, memory locations, and instructions (as illustrated in Figure 3). As is shown in the figure, data dependencies are tracked across registers as well as memory locations. Also, control dependencies between basic blocks are shown. Note that the resulting graph is directed and acyclic since dependencies can only “point downward” (i.e., no instruction can ever depend on a later instruction).

The resulting dependency graph considers all available parallelism in the application. Of course, it does not and cannot consider any potential parallelism that could be achieved by choosing a different instruction set or different algorithms and data structures for the application. Finding such parallelism is beyond the scope of our work.

3.3 ADAG Generation

When mapping processing steps on a network processor architecture, it needs to be considered that there is a tradeoff between the cost of processing and the cost of communication. This implies that it is not desirable to parallelize small processing blocks on multiple processors, especially when this requires a large amount of communication.

The initial trace from PacketBench results in a graph with thousands of basic blocks. The dependencies between them can cause a large amount of communication if the basic blocks were to be distributed to different computational resources. Thus, we want to reduce the number of processing tasks to yield a more natural, tractable grouping of processing instructions. For this purpose, we use a clustering technique called “ratio cut” [Wei and Cheng 1991]. 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 :

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

A clustering algorithm that minimizes r_{ij} 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 of exponential complexity 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 based on the ratio cut metric and reduces the computational complexity while still achieving good results [Ramaswamy et al. 2004]. The idea of MLRC is to greedily find the pair of clusters that maximize r_{ij} (and thus show most cohesiveness) and join them.

While an entirely “natural” clustering of instructions would represent the application most accurately, it might not yield an ADAG that is suitable for mapping on a network processor. Since we are considering pipelined NP architectures, we need to keep in mind that pipeline performance is determined by the slowest stage. This means that ADAGs with clusters that vary considerably in size are difficult to map. To avoid this problem, MLRC is tuned to bound the ratio between the largest and the smallest processing task.

3.4 Sample Workload and ADAGs

To illustrate the results of the ADAG generation process, we consider a workload consisting of four typical network processors applications. These applications are the basis of evaluation for the remainder of this paper. They are:

- IPv4-radix.** IPv4-radix is an application that performs RFC1812-compliant packet forwarding [Baker 1995] 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. The depth of the structure increases very slowly with the number of entries in the routing table. More details can be found in [Nilsson and Karlsson 1999].
- 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 [Daemen and Rijmen 2000], which is used in the Advanced Encryption Standard (AES).

After simulating these applications, collecting run-time traces, identifying data and control dependencies, and clustering instructions to processing tasks, we obtain the ADAGs shown in Figure 4. 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. The amount of 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 in terms of parallel and sequential behavior. IPv4-radix and IPSec are highly sequential applications. Both applications run a loop over several iterations (different nodes in the ADAG). Due to data

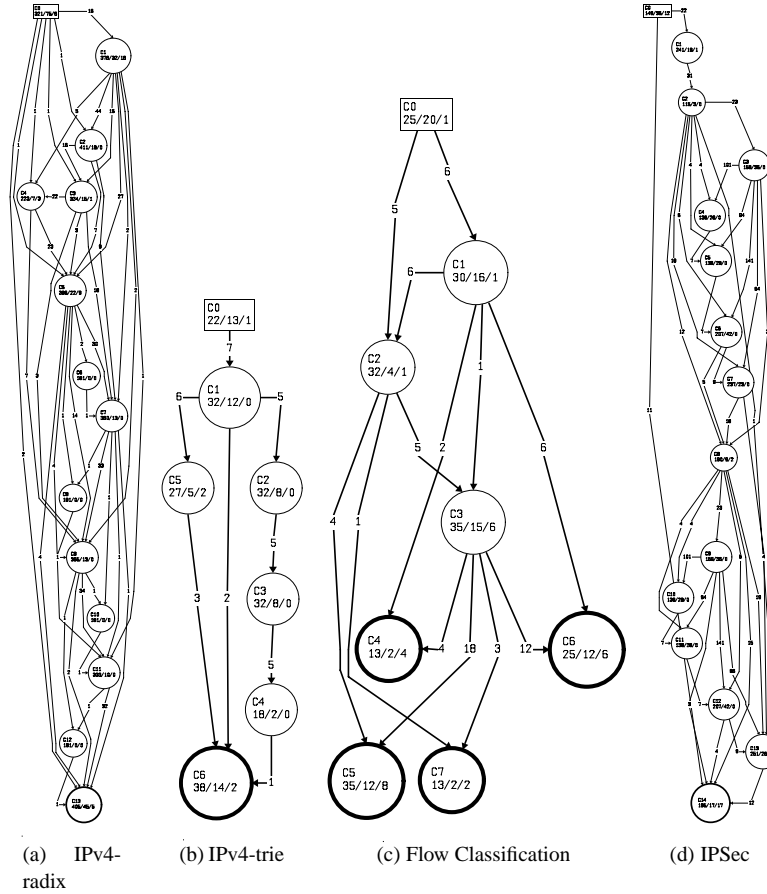


Fig. 4. Annotated Directed Acyclic Graphs (ADAGs) for Workload Applications. The annotation in a node is the node name and a 3-tuple (processing/reads/writes). The weight on the edges is the number of data and control dependencies between nodes.

dependencies (lookup result of previous radix node or cryptographic result of previous iteration) the overall ADAG is entirely sequential. IPv4-trie and Flow Classification display more parallelism in their ADAGs.

3.5 Applications and Hardware Acceleration

The final question in the context of application analysis is how to identify processing blocks that lend themselves for hardware acceleration on the heterogeneous network processor system (e.g., co-processor or programmable logic). In principle, there are two approaches to solving this problem:

- Manual Assignment of Code Blocks.** The application developer can mark functions or code blocks that can be implemented using co-processors. This requires that the developer be aware of the hardware accelerators that are available on the system.
- Automatic Identification of Repetitive Processing.** An approach that does not involve

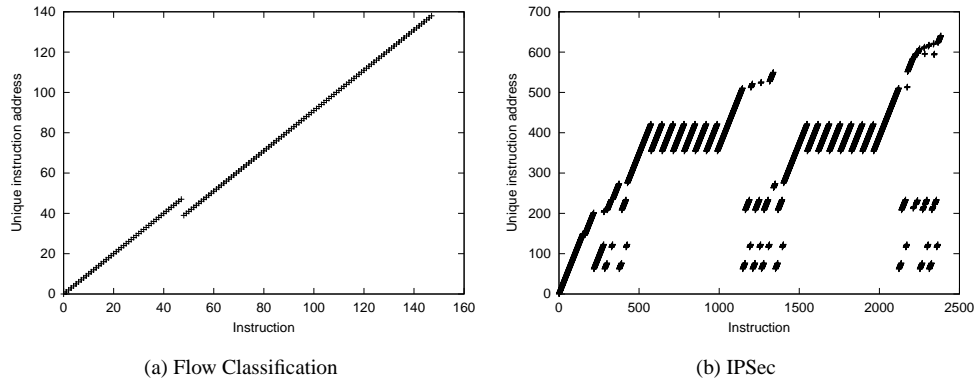


Fig. 5. Detailed Instruction Access Patterns of a Single Packet for Flow Classification and IPSec from the MRA Trace.

the input of the application developer is to use the run-time trace to identify repetitive processing. Frequently reoccurring instruction addresses are a good indicator for loops. If such a loop is a processing-intensive component of the application, then it might be desirable to accelerate this portion of the application to increase overall throughput.

As we have stated earlier, our goal is to develop an analysis and mapping methodology that can operate automatically and shields the developer from the details of the system architecture. Thus, we pursue the second approach. It is clear that such an approach cannot identify all possible co-processing functions, but we want to take a different look at the problem and attempt to identify such functions without a-priori understanding of the application.

The ADAGs only show how many instructions are executed by a processing block, but not which instructions. In order to identify if there are instruction blocks that are heavily used in an application, we use the plots shown in Figure 5. The x-axis represents the instructions that are executed during packet processing. The y-axis shows each unique instruction address observed during packet processing. For example, in IPSec, the 400th unique instruction is executed sixteen time (eight times between instruction 500 and 1000 and eight times between 1500 and 2000).

Figure 5 is a good indicator for repetitive execution of the same instruction addresses. For Flow Classification, there are almost no repeated instruction addresses. In IPSec, however, there are several instruction blocks that are executed multiple times (sixteen times for instructions with unique address 350 to 450). If these instructions can be implemented in dedicated hardware, a significant speed-up can be achieved due to the high utilization of this block.

Again, this method of co-processing identification requires no knowledge or deep understanding of the application. Instead the presented methodology extracts all this information from a simple instruction run-time trace. One problem with this methodology is that processing blocks that execute non-repetitive functions are not identified as suitable for co-processors, even though they could be (as it is the case for Flow Classification). Such functions still need to be identified manually by the programmer.

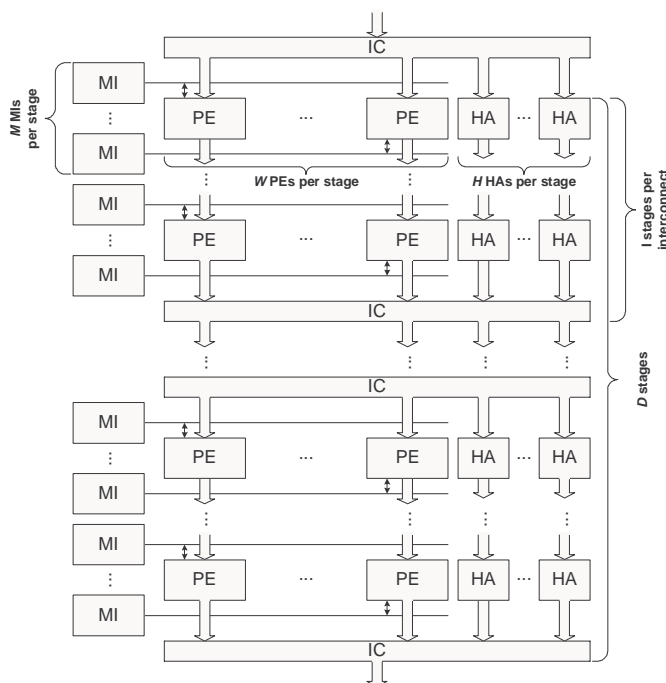


Fig. 6. Generalized Network Processor Architecture. The parameters W , D , M , I , and H determine the number and topology of processing elements (PE), interconnects (IC), memory interfaces (MI), and hardware accelerators (HA).

The hardware acceleration functions are represented in an ADAG as a specially annotated group of nodes because—if an accelerator is used—they have to be mapped in their entirety to the hardware accelerator. It should be noted that of course the accelerator does not perform the actual RISC instructions that were obtained from the instruction trace. Nevertheless they are represented as such in the ADAG in order to allow an accurate modeling of the potential use of accelerators as discussed in Section 5.

4. WORKLOAD MAPPING

With applications represented by ADAGs we are able to map the workload to the network processors. The throughput performance of the system not only depends on the workload and the mapping, but also the underlying NP architecture.

Network processors exhibit a large diversity in architecture—with system topologies ranging from simple multiprocessors to complex data flow pipelines. Instead of limiting the results to one particular commercial NP architecture, we use a parameterized network processor architecture that captures the major system aspects of any network processor while abstracting away implementation details. The mapping algorithm is designed to allocate workloads on all possible NP configuration.

4.1 Parameterized Network Processor Architecture

Our general, parameterized network processor topology, shown in Figure 6, consists of four components: processing elements (PE), shared interconnects (IC), memory interfaces

(MI), and hardware accelerators (HA). The packets move from top to bottom. The key parameters are: the width of the pipeline (W), the depth of the pipeline (D), the number of stages per communication interconnect (I), the number of memory channels shared by a stage of processing elements (M), and the number of hardware accelerators (H) per stage.

These parameters enable us to represent a wide range of possible NP architectures:

- Parallel Multiprocessor Topology.** This topology can be modeled by fixing the pipeline depth (D) and number of stages per interconnect (I) to 1, while varying the pipeline width (W). A commercial example for this topology is Intel’s IXP2400 ($W=8$ and $H=1$ for a hash co-processor).
- Pipelined Architecture.** A pipeline architecture can be modeled by setting both pipeline width (W) and number of stages per interconnect (I) to 1, while varying the pipeline depth (D). Agere’s Fast Pattern Processor and Routing Switch Processor is an example for such a topology ($D=2$).
- Hybrid Architecture.** A hybrid architecture can be achieved by setting $W, D, I=1 \dots D$, and $M=1 \dots W$ to any combination of values. EZchip’s NP-1 uses such a topology ($W=4, D=4, I=1$).

For simplicity in the discussion, we treat hardware accelerators as if they could perform any hardware acceleration task that was identified during the application analysis. Of course, realistic systems have limitations on which task can be performed on which accelerator and each accelerator $HA_{1,1} \dots HA_{D,H}$ would use a mapping $H_{i,j} \rightarrow T_{i,j}$, where $T_{i,j}$ is the set of tasks T that can be accelerated.

4.2 Mapping

With a workload of multiple application ADAGs and a parameterized network processor topology as inputs, the mapping algorithm determines the assignment of processing tasks (i.e., ADAG nodes) to processing elements. The goal of the mapping is to generate an assignment that achieves the maximum system throughput.

In our work, we focus on mapping tasks to processors, but in an operational system, this mapping need to be translated into a schedule by determining *when* a packet should be processed. Using the ADAG abstraction and the generalized network processor architecture, a mapping can easily be translated into a schedule. The top node of an ADAG is the starting point for packet processing and whenever that processing task is idle, a new packet can be assigned to this ADAG. The synchronization across pipeline stages ensures that lower nodes in the ADAG have completed processing by the time the next packet is passed on from a higher node. As a result, each ADAG can start (and in steady-state complete) the processing of one new packet per step of the pipeline. We assume that the workload of the system is known a priori (e.g., how many packets need IP forwarding and how many need IPSec processing). Thus we can map the appropriate ratio of ADAGs from each application to the system and ensure that all ADAGs are fully utilized at all times. In an operational network node, the run-time system would adjust the proportions of application ADAGs dynamically as traffic requirements change [Wolf et al. 2005], but this dynamic scenario goes beyond the scope of this paper. Instead, we focus on finding the best mapping for a static workload.

The assignment of processing resources 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 in 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 in [Malloy et al. 1994] 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. Therefore we need to develop a heuristic to find an approximate solution.

4.3 Randomized Mapping

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 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 [Karp 1991], Motwani and Raghavan [Motwani and Raghavan 1995], and Lakamraju et al. [Lakamraju et al. 2002].

Our randomized mapping algorithm is shown as Algorithm 1. We break the mapping algorithm into two stages: mapping and filtering. In the mapping stage (lines 1–18), we randomly allocate an ADAG node (i.e., a processing task) to a processing element in the NP topology. The mapping traverses the entire DAG (lines 12–18) and places each node (lines 1–11). To avoid node placements that lead to an illegal configuration, any given ADAG node can only be mapped to the same processing element as its parent (i.e., (i, j) ¹) or to any processing element in stages after that of the parent (i.e., (i', j') with $i' > i$). Thus, the placement is chosen randomly (line 5) from valid choices among three cases: (1) either the node is placed with its parent (line 6) or (2) it is placed on a general-purpose processor (line 7) or (3) it is placed on a general-purpose processor or hardware accelerator (line 8). In each of those cases, the function HA is used to determine if a node is or can be placed on a hardware accelerator.

The mapping result is evaluated in the filtering stage (lines 19–31) where its performance is compared to the best prior mapping. If the new mapping is a better solution in terms of the optimization metric, it is recorded for comparison to future solutions (lines 26–28). Otherwise it is discarded. This mapping and filtering process is repeated for a set number of attempts (lines 21–29). At the end of the mapping process, the best overall mapping is reported (line 30). It is possible that the mapping may place multiple ADAGs (line 23)

¹The pair (i, j) indicates the location of the processing element in the NP architecture. The stage is identified by i and the processor within the stage by j . The value of j identifies general-purpose processing when $1 \leq j \leq W$ and hardware acceleration when $W+1 \leq j \leq W+H$

```

1 RandomPlacement (n)
2 begin
3   p ← parent(n);
4   (i,j) ← CommittedPlacement (p)
5   switch randomly from valid choices do
6     (i',j') ← (i,j) if (HA (p) ∧ HA (n)) ∨ (¬HA (p) ∧ ¬HA (n))
7     (i',j') ← (Rand (i + 1 ... D), Rand (1 ... W)) if ¬HA (n)
8     (i',j') ← (Rand (i + 1 ... D), Rand (1 ... W + H)) if HA (n)
9   end
10  CommitResources (map,n,(i',j'));
11 end
12 MapNode (n)
13 begin
14   RandomPlacement (n);
15   foreach n' in NotYetMappedChildren (n) do
16     MapNode (n');
17   end
18 end
19 RandomizedMapping ()
20 begin
21   for attempts ← 1 to maxAttempts do
22     map ← null;
23     foreach ADAG a in workload do
24       MapNode (TopNode (a));
25     end
26     if Performance (map) > Performance (bestMap) then
27       bestMap ← map;
28     end
29   end
30   return bestMap;
31 end

```

Algorithm 1: Randomized Mapping Algorithm

onto the network processor (either different ADAGs to reflect a workload mix or multiple instances of the same ADAG to utilize more resources for one application), which means that packets can be processed in parallel.

The complexity of one iteration of the randomized mapping algorithm is $\mathcal{O}(nm)$, where n is the number of nodes in an ADAG, and m the number of ADAGs. The complexity is independent of the number of processing elements p , but typically there are more nodes than processing elements ($nm > p$) in order to fully utilize the system.

Figure 7 shows the range of solutions that are derived during randomized mapping. The x-axis shows the iteration of the mapping algorithm. The y-axis shows 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 have 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

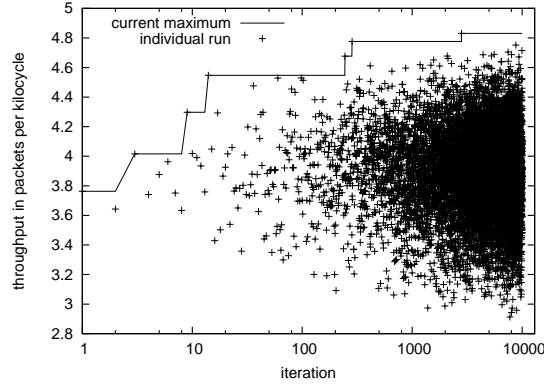


Fig. 7. Randomized Mapping. Performance of randomized mapping result over 10,000 iterations.

possible solutions. This would not be possible if more time consuming simulation-based performance evaluation methods were used.

5. PERFORMANCE MODEL

The performance model that is used in the mapping process provides an analytic expression for the system throughput. After mapping the application ADAGs to the network processor topology, we know the exact workload of each processing element. This information is derived from the ADAGs and includes the total number of instructions executed, the number of memory accesses, and the amount of communication between stages. The model uses this information to determine processing times as well as contention for shared resources (memory channels and communication interconnects).

The high latency of remote memory accesses is a major impediment to good application performance on scalable multiprocessors. Several techniques have been proposed for coping with this problem. One of them is multithreading, which tries to overlap the long memory access delay for one thread with computations in other threads to reduce processor stalls. We first introduce the performance model for a single thread system and then expand the model to multithreading.

5.1 Single Thread Model

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, $\tau_{stage_{max}}$, because the critical stage of the pipeline architecture determines the overall system speed due to the synchronized nature of pipelined architecture. The number of ADAGs that are mapped to an architecture, n_{ADAG} , determines how many packets are processed in parallel during a pipeline stage. The throughput of the system is expressed as

$$throughput = \frac{n_{ADAG}}{\tau_{stage_{max}}}, \quad (2)$$

where the maximum stage time is the maximum time that any of the D stages spends on processing, $\tau_{proc_{i,j}}$, and memory accesses, $\tau_{mem_{i,j}}$, (or hardware acceleration, $\tau_{HA_{i,k}}$.)

and communication, τ_{comm_i} :

$$\tau_{stage_{max}} = \max_{i=1}^D \left(\tau_{comm_i} + \max \left(\max_{j=1}^W (\tau_{proc_{i,j}} + \tau_{mem_{i,j}}), \max_{k=1}^H (\tau_{HA_{i,k}}) \right) \right). \quad (3)$$

For a general purpose RISC processor with single-cycle instructions, the processing time, $\tau_{proc_{i,j}}$, depends solely on the number of instructions, $instr_{i,j}$ that are mapped to this processing element during the mapping phase:

$$\tau_{proc_{i,j}} = instr_{i,j} \quad (4)$$

Due to the stringent constraints on worst-case performance of practically all networking applications, commonly used algorithms and their data structure exhibit a complexity between $\mathcal{O}(1)$ and $\mathcal{O}(n)$. Therefore we assume that hardware accelerators that implement these algorithms require either an amount of time that is constant (e.g., for hash computation) or that is linear to the amount of data that is processed (e.g., checksum computation). Since modeling more complex processing behavior of hardware accelerators is beyond the scope of this work, we assume that $\tau_{HA_{i,k}} = \alpha$ or $\tau_{HA_{i,k}} = \beta \cdot instr_{HA_{i,k}}$, where α and β are characteristics of the hardware accelerator and $instr_{HA_{i,k}}$ are the general-purpose instructions that are replaced by hardware accelerator $HA_{i,k}$.

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 modeled as a Machine Repairman model with a fixed number of sources (i.e., processing elements) and a certain number of servers (i.e., memory channels). There is no closed form solution for the Machine Repairman system. Instead the result is computed iteratively over the number of job servers. The response time for a system with n job servers is [Reijns and van Gemund 1999]

$$R_n = S + SQ_{n-1} - \frac{S}{2}U_{n-1}[1 - U_{n-1}][\frac{2}{S}R_{n-1} - 1], \quad (5)$$

where S is the service time of the memory interface (i.e., memory access time), U_{n-1} is the utilization of the memory interface when there are $n - 1$ processing elements active in the system, and Q_{n-1} is the number of requests present at the memory interface (including the current one served), if there are $n - 1$ processing elements active in the system.

With the response time R_n , the number of memory read $r_{i,j}$ and write $w_{i,j}$, the overall memory access time for a processing element can be determined as

$$\tau_{mem_{i,j}} = R_n \cdot (r_{i,j} + w_{i,j}). \quad (6)$$

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

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. The amount of data that is communicated across an interconnect $1 \leq k \leq D$, $data_k$, is:

$$data_k = \sum_{i \leq k, 1 \leq j \leq W} (dep_{(i,j),(i',j')} | k < i' \leq D \wedge 1 \leq j' \leq W), \quad (7)$$

where $dep_{(i,j),(i',j')}$ is the amount of data transferred from processor (i, j) to processor (i', j') as determined by the edge weight of the mapped ADAGs. Assuming each data

element can be communicated in one clock cycle, the stage communication cost is:

$$\tau_{comm_k} = data_k. \quad (8)$$

Equations 4, 6, and 8 can be substituted into Equation 3 to obtain the overall system throughput. The throughput determines the quality of a mapping on a particular architecture as shown in Figure 7.

5.2 Multi-threaded Model

The throughput of the multi-threaded system is similar to the single thread model given in Equation 2. With t threads operating on a processor, t times as much work gets done, but the stage time, $\tau_{stage_{max}}(t)$, increases:

$$throughput = t \cdot \frac{n_{ADAG}}{\tau_{stage_{max}}(t)}. \quad (9)$$

For multithreaded models, we need to distinguish between the cases as discussed in [Agarwal 1992]: (a) processing performance is bound by processing speed (i.e., memory is fast enough that multithreading can hide latencies) and (b) processing performance is bound by memory accesses (i.e., memory is not fast enough and processor stalls occur). The equation for the stage time is similar to 3, however with these two cases and the context switching overhead C considered, $\tau_{stage_{max}}(t)$ is:

$$\tau_{stage_{max}}(t) = \begin{cases} \max_{i=1}^D \left(t \cdot \tau_{comm_i} + \max \left(\max_{j=1}^W (t \cdot (\tau_{proc_{i,j}} + C)), \max_{k=1}^H (\tau_{HA_{i,k}}) \right) \right) & \text{(a)} \\ \max_{i=1}^D \left(t \cdot \tau_{comm_i} + \max \left(\max_{j=1}^W (\tau_{proc_{i,j}} + \tau_{mem_{i,j}}(t)), \max_{k=1}^H (\tau_{HA_{i,k}}) \right) \right) & \text{(b)} \end{cases} \quad (10)$$

While the communication time τ_{comm_i} and processing times $\tau_{proc_{i,j}}$ and $\tau_{HA_{i,k}}$ are the same as in the single-threaded model, the memory access time $\tau_{mem_{i,j}}$ is a function of the number of threads t :

$$\tau_{mem_{i,j}}(t) = R_n(t) \cdot (r_{i,j} + w_{i,j}). \quad (11)$$

The response time for a system with n job servers and t threads is:

$$R_n(t) = S + SQ_{n-1}(t) - \frac{S}{2} U_{n-1}(t) [1 - U_{n-1}(t)] \left[\frac{2}{S} R_{n-1}(t) - 1 \right] \quad (12)$$

The response time is calculated iteratively as in the single-threaded model.

With the analytic performance model, the throughput for any given mapping solution can be determined. This result is used in the randomized mapping algorithm to determine the overall best mapping solution.

6. RESULTS

We present several results that show the performance of the task allocation mechanism discussed above. The validity and correctness of our results is established in three steps: First, we validate the mapping results for a simple scenario by comparing them to theoretical upper bounds. Second, for a more complex scenario, we show the effectiveness of randomized mapping by comparing it to exhaustive search. Third, for the validation of our

analytic performance model, we compare the obtained results with those of a commercial NP simulator.

We also present results that compare the throughput performance of different network processor topologies. We show how the limitations in instruction store on realistic NPs impacts these results. Finally, we evaluate different NP architecture for their suitability for different applications.

The results in this section were obtained for all four applications discussed in Section 3. Due to space limitations, the shown results are limited to a representative subset.

6.1 Mapping Speed and Correctness

As shown in Figure 7, the randomized mapping algorithm quickly obtains a feasible mapping and then discovers better solutions with an increasing number of iterations. For a practical network processor run-time system, this feature is particularly important because processing power is limited and frequent re-mapping might occur due to changing network conditions. In our simulations, we used an un-optimized implementation of the mapping algorithm and the performance model. On a 2.4GHz Pentium processor, one iteration of randomized mapping for a 16-processor topology can be computed in 12ms. It should also be noted that the mapping algorithm can be adapted to support incremental changes (i.e., removal of one ADAG and adding of another).

Any mapping algorithm needs to yield correct results in order to be useful. Correctness requires that entire applications are mapped and data and control dependencies are considered. Our randomized mapping methodology yields correct mappings by design, because every dependency is considered in the mapping step. Not every correct solution is necessarily efficient and the mapping quality is of major importance.

6.2 Mapping Quality

To illustrate the quality of the randomized mapping heuristic, we compare our mapping results to the theoretically optimal mapping. As pointed out earlier, the mapping problem is NP-complete and thus we need to limit ourselves to a simple scenario for this comparison. We consider a simple multiprocessor topology ($D=1$) with no hardware accelerators, for which we can derive an analytic expression for the optimal processing performance. In the ideal case, all processing elements execute the same number of instructions and memory accesses. Communication is limited to the initial input and final output. The ideal throughput can then be derived from the equations presented in Section 5 by substituting average processing and memory access values. The optimal throughput is:

$$throughput_{ideal} = \frac{1}{\frac{instr_A}{W} + mem_{delay} \cdot (r_A + w_A)}. \quad (13)$$

The throughput is independent of the number of ADAGs because processing and memory accesses are averaged out and $\frac{n_{ADAG}}{W}$ instructions are allocated to each processing element.

Table I shows a comparison between the optimal throughput and the throughput that is achieved by our randomized mapping approach ($W=8$). The difference is around 1% due to application fragmentation. Only complete ADAG nodes can be allocated to processing elements and thus the applications cannot be distributed evenly as in the theoretical case. This shows that randomized mapping can achieve a near-optimal application mapping for

Table I. Optimal Mapping vs. Randomized Mapping Throughput for Different Applications.

Application	Instructions	Memory accesses	Optimal throughput in pkts/kcycl	Randomized 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%

Table II. Mapping of Applications With and Without Hardware Acceleration.

Application	General-Purpose Architecture ($D=4, W=4, H=0$) throughput in pkts/kcycl	Architecture with HA ($D=4, W=3, H=1$) throughput in pkts/kcycl
IPv4-radix	0.70	0.73
IPv4-trie	2.45	2.82
Flow Class.	1.40	1.60
IPsec	0.31	0.39

this topology.

Table II compares the performance achieved by the mapping of applications on two NP architectures – one with general-purpose processors and one where one processor per stage is replaced by a hardware accelerator. For the experiment, one node of each ADAG was identified to be suitable for hardware acceleration, where it could be processed at approximately twice the speed of a general purpose processor. The results show that our mapping algorithm correctly considers hardware accelerators for mapping to achieve better performance.

6.3 Comparison to Exhaustive Search Algorithm

The disadvantage of the upper bound derived in Equation 13 is its applicability to pool topologies (pipeline depth $D=1$) only. For any other topology, it is very difficult, if not impossible, to identify the best mapping and to derive the upper bound of the system performance. To obtain some basis for comparison for such topologies, we apply an exhaustive search (or “brute force”) algorithm. The basic idea of exhaustive search is to enumerate all the possible mappings and identify the one that produces the best performance. The complexity of exhaustive search is $\mathcal{O}(p^{nm})$, where p is the number of processing elements, n the number of nodes in an ADAG, and m the number of ADAGs. Due to the exponential growth in complexity, only relatively small topologies and ADAG configurations can be explored. For some of those configurations, we have compared our randomized mapping results to those of exhaustive search and found that randomized mapping does find solutions with the same performance as does exhaustive search. Also, in most cases randomized mapping finds these solutions in a fraction of the iterations that are necessary for exhaustive search (e.g., for a hybrid topology ($D=2, W=3$) with three 5-node ADAGs, randomized mapping finds the optimal solution in 1% of the iterations of exhaustive search).

Table III. Application (4Gb Ethernet IPv4 ingress) Partition and Characterization. Each row of the table corresponds to one of functional blocks of the application partition.

Processing engine	Number of instructions	Number of SRAM accesses	Number of SDRAM accesses	Number of ScratchPad accesses
0	66	1	1	1
1,2,5,6	72	9	1.8	5
3	74	2	0	4
4	74	0	0	1
7	81	2	1	1.5

6.4 Performance Model Validation

To validate the analytic performance model that we have developed above, we compare its performance results with those obtained from a cycle-accurate system simulator. We use the Intel DeveloperWorkBench [Intel Corporation 2003], which simulates an Intel IXP2400 network processor. The IXP2400 system contains eight multi-threaded processing engines and different types of memory (registers, on-chip SRAM, off-chip SRAM, and off-chip SDRAM). Since this memory configuration is somewhat different from our generalized network processor architecture shown in Figure 6, we adapt the memory model accordingly.

For comparison, we consider two applications: 4Gbps IPv4 Ingress processing for Ethernet and OC-48 POS MPLS Ingress processing. The partitioning of tasks is such that one processing engine is allocated to RX Processing, Queue Manager, CSIX Transmit Scheduler, and CSIX Transmit, respectively. The remaining four processing engines are allocated to either of the two applications. Table III shows the partition and workload characterization for 4Gbps IPv4 Ingress. The partition for OC-48 POS MPLS Ingress is similar (not shown). The table also shows the number of memory accesses separated by type of memory. Fractional memory accesses are caused by averaging over different execution paths.

The key metric required for the model-based performance estimation is the maximal stage time. Here the maximum stage time is the maximum time that any of the micro-engines spends on packet processing. The maximal stage time $\tau_{stage_{max}}(t)$ is calculated by Equation 10. The memory access time can be calculated by Equation 6. However, instead of a single memory access time, we distinguish the response time R_n for each of the tree types.

The throughput performance is measured with a simulation of 100,000 packets (size 68 bytes). Table IV shows a comparison between the estimated throughput by the analytic model and the throughput that is obtained from the simulator. The differences are around 1% for one application and around 8% for the other. This shows that the analytic model derives results that are quite close to cycle-accurate simulation results.

This and the previous two results show that our proposed methodology of workload mapping and performance modeling achieves correct results and performance estimates that are very close to simulated results. With correctness established, we turn to exploring system-level design choices.

Table IV. Analytic Performance Model vs. Cycle-accurate Simulation

Application	Model-based throughput in Mbps	Simulator-based throughput in Mbps	Difference in %
IPv4 Ingress	4,014	4056	1.04%
MPLS Ingress	2,293	2,501	8.32%

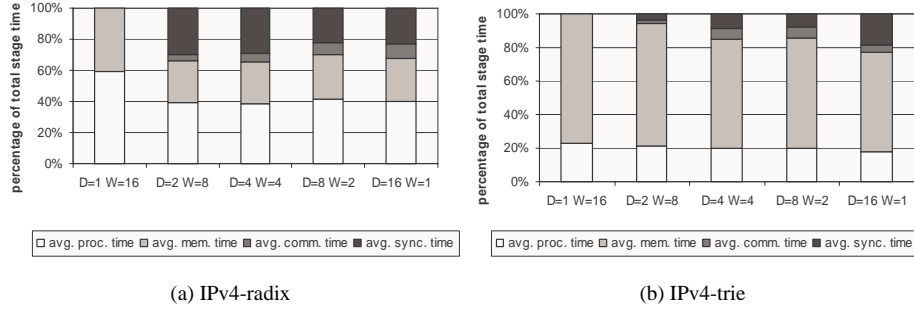


Fig. 8. Composition of Stage Time for Different Topologies. The memory service time is $S=10$, topologies range from parallel ($D=1, W=16$) to pipelined ($D=16, W=1$), and there is only one thread per processor ($t=1$).

6.5 Topology Impact on Throughput Performance

To illustrate the generality of our proposed methodology, we explore the throughput performance of different applications on a range of different network processor topologies. We vary the NP architecture from a fully parallel to a fully pipelined system including several hybrid configurations in between. In all cases the total number of processing elements and memory channels is kept constant.

In addition to processing and memory access time, we also observe communication and synchronization overhead in pipelined topologies. The communication overhead is caused by moving data between processing stages. The synchronization overhead is due to the enforced synchrony in the pipeline, which is the difference between the maximal stage time and current stage time. The communication overhead can be reduced by clustering the ADAG well and allocating multiple nodes from one ADAG on the same processing element. The synchronization overhead can be reduced by finding better task allocations that assign equal amounts of processing and memory accesses to each processor.

Figure 8 shows the composition of the stage time of the IPv4-radix and IPv4-trie applications on different topologies ranging from parallel ($D=1, W=16$) to pipelined ($D=16, W=1$), each with a total of 16 memory channels. The number of threads per processor is $t=1$ and no hardware accelerators are considered. The stage time is normalized to 100% to simplify comparison. The actual throughput performance of each configuration is shown in Figure 9 in the following section.

Overall, processing and memory access contribute most to the overall stage time. For architectures with pipelining ($D>1$), synchronization and communication add to the overall

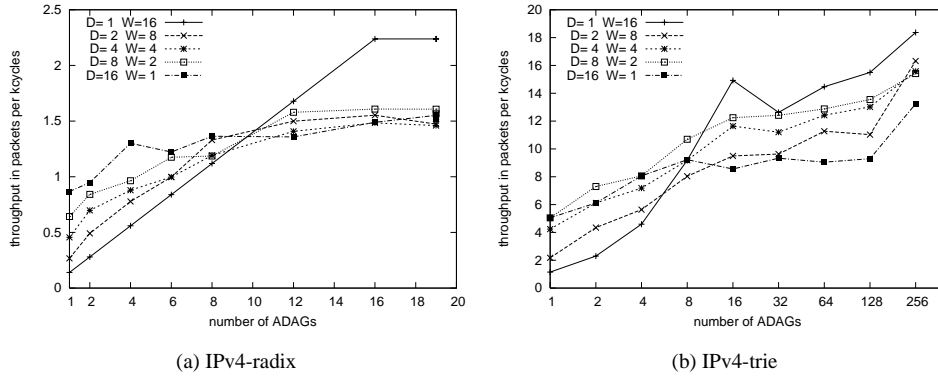


Fig. 9. Number of ADAGs Depending on Topology. The memory service time is $S=10$ and there is only one thread per processor ($t=1$).

stage time. This is the cost for using pipelining in a system. For IPv4-radix, a small number of ADAGs ($n_{ADAG} < 16$) is mapped and utilization across processors is less balanced than for IPv4-trie, where a large number of ADAGs ($n_{ADAG} > 200$) is mapped. Large numbers of ADAGs helps the mapping algorithm to evenly distribute processing tasks and reduce the synchronization overhead. In a practical system, however, the number of ADAGs that can be mapped is limited by the available instruction store per processing element, which is explored in the following section.

6.6 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). In many implementations, this instruction store is local to a processing engine, does not use caching, and cannot be shared with other processing engines. While the *static* instruction store does not directly translate into the *dynamic* instructions shown in Table I, these constraints imply that there is a limit to how many ADAG nodes can be mapped to a particular processing element. If the mapping assigns the same nodes from different instances of the same ADAG to the same processing element, then instructions can be reused. Since this is a very special case, we do not consider it further and assume that each ADAG node requires its own instruction store. Therefore it is important to consider the tradeoff between the need for mapping many ADAGs to achieve balanced processor utilization (Figure 8) with the limitations of instruction store.

Figure 9 shows the throughput performance of the two applications for different number of ADAGs (x-axis) for different architectures. The key observation is that for a small number of ADAGs (i.e., low instruction store requirements) the pipelined architectures perform better. For larger numbers of ADAGs (i.e., high instruction store requirements) 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 mapped 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 ar-

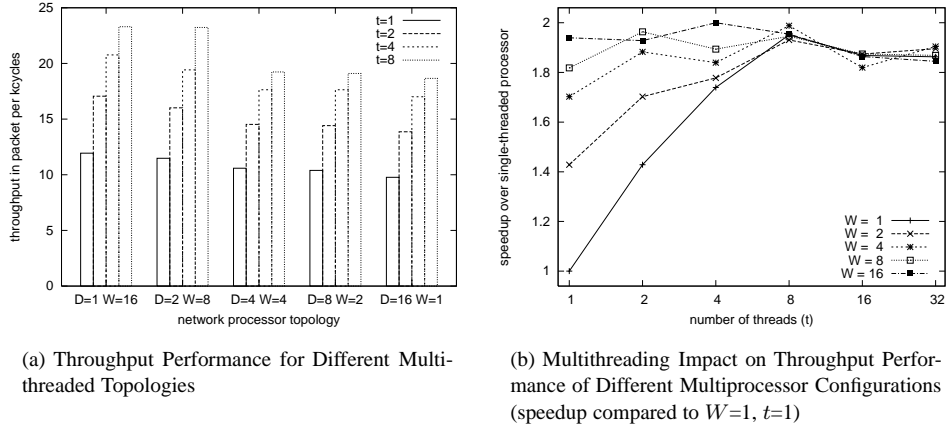


Fig. 10. Performance Tradeoffs of Multithreading Configuration. Results are for the IPv4-trie application and a total of $D \cdot M=16$ memory channels.

architectures perform better for small numbers of ADAGs. With an increasing number of ADAGs communication between processing elements increases and limits the throughput.

For a network processor designer this means that pipelined architectures can only be fully utilized if applications can be partitioned such that synchronization and communications do not pose larger overheads. This can be achieved by slowing down the pipeline and allocating more processing to each stage (as shown in Figure 9) or by possibly removing the constraint of synchrony in the pipeline.

6.7 Network Processor Topology Design Space 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 mapping and performance model methodology to explore this issue quantitatively in more detail.

We have modeled the throughput performance of NP topologies with configurations of $D=1 \dots 16$ and $W=1 \dots 16$ (figure not shown). The memory channels are set to $M=2$ per stage and there is one thread per processor. For increasing pipeline width, there is limited growth in performance due to contention on the two memory channels per stage. Also the off-chip memory access time of $S=10$ causes memory accesses to dominate the overall stage time. For all applications, the performance increases as the pipeline depth increases. Some application levels off at larger number of pipeline stages (e.g., IPsec at depth 8). 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.

6.8 Results Summary

These results lead to two conclusions regarding NP topologies. Memory accesses dominate as performance bottlenecks and pipelining requires large instruction stores to allow

for a balanced mapping. Of course, multithreading is a straightforward solution to reducing the impact of memory latency. Figure 10(a) shows that an increasing number of threads improves throughput performance for any topology. It is important to note, however, that multithreading is only useful in conjunction with sufficient memory bandwidth. Figure 10(b) shows a comparison the performance improvement of a multithreaded system in relation to a single threaded system. The increasing width of the topology ($D=1$ and $W=1 \dots 16$) reflects increasing contention on the memory system, which poses as bound on the maximum throughput performance. Thus, a large number of threads (as implemented in many current commercial NPs) need to be supported by a fast, high-bandwidth memory system in order to have any impact on system throughput.

While the conclusions from these results are not unexpected, they support two important observations. First, the mapping and modeling approach that we present yields intuitively correct answers, and second, the model allows us to quantify the trends to make specific decisions on network processor design and programming.

7. CONCLUSION

In this work, we have introduced a methodology for profiling and mapping networking workloads on highly parallel network processor architectures. The mapping is based on a randomized heuristic that achieves a good approximation to solve this NP-complete problem. The analytic performance model captures the key system aspects of a heterogeneous network processor system and allows an easy evaluation of application mappings on a wide range of network processor configurations. The results show the quality of the mapping results and discuss the various tradeoffs between network processor topologies in the context of instruction store limitations. We also present how the analytic performance model can be applied for understanding tradeoffs in the NP design space to determine suitable network processor topologies and multithreading configurations.

We believe that this methodology poses a promising approach to managing the complexities of highly parallel, heterogeneous network processors and embedded systems in general. The profiling, mapping, and performance evaluation can be done entirely automatically from a uniprocessor implementation of an application. It is conceivable that such functionality will become part of software development kits and run-time environments of future network processors.

REFERENCES

- AGARWAL, A. 1992. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems* 3, 5 (Sept.), 525–539.
- AUSTIN, T. M. AND SOHI, G. S. 1993. Tetra: evaluation of serial program performance on fine-grain parallel processors. Tech. Rep. 1163, Computer Science Department, University of Wisconsin, Madison, WI. July.
- BAKER, F. 1995. Requirements for IP version 4 routers. RFC 1812, Network Working Group. June.
- BHANDARKAR, D. P. 1975. Analysis of memory interference in multiprocessors. *IEEE Trans. on Computers* c-24, 9 (Sept.), 897–908.
- DAEMEN, J. AND RIJMEN, V. 2000. The block cipher Rijndael. In *Lecture Notes in Computer Science*. Vol. 1820. Springer-Verlag, 288–296.
- DOWDY, L. W., ROSTI, E., SERAZZI, G., AND SMIRNI, E. 1999. Scheduling issues in high-performance computing. *SIGMETRICS Performance Evaluation Review* 26, 4 (Mar.), 60–69.
- FOSTER, I. AND KESSELMAN, C., Eds. 2004. *The Grid – Blueprint for a New Computing Infrastructure*, 2nd ed. Morgan Kaufmann.

- FRANKLIN, M. A. AND WOLF, T. 2002. A network processor performance and design model with benchmark parameterization. In *Proc. of First Network Processor Workshop (NP-1) in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*. Cambridge, MA, 63–74.
- FRANKLIN, M. A. AND WOLF, T. 2003. Power considerations in network processor design. In *Proc. of Second Network Processor Workshop (NP-2) in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*. Anaheim, CA, 10–22.
- GOGLIN, S. D., HOOPER, D., KUMAR, A., AND YAVATKAR, R. 2003. Advanced software framework, tools, and languages for the IXP family. *Intel Technology Journal* 7, 4 (Nov.), 64–76.
- GRASSO ET AL., P. A. 1984. Memory interference in multimicroprocessor systems with a time-shared bus. *Proceedings of the IEEE* 131, 10 (Mar.).
- GRIES, M., KULKARNI, C., SAUER, C., AND KEUTZER, K. 2003. 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)*. Anaheim, CA, 75–87.
- HOOGENDOORN, C. H. 1977. A general model for memory interference in multiprocessors. *IEEE Trans. on Computers* c-26, 10 (Oct.), 998–1005.
- Intel Corporation 2003. *Intel IXA Software Developers Kit 2.01*. Intel Corporation.
- KAPASI, U. J., RIXNER, S., DALLY, W. J., KHAILANY, B., AHN, J. H., MATTSON, P., AND OWENS, J. D. 2003. Programmable stream processors. *IEEE Computer* 36, 8 (Aug.), 54–62.
- KARP, R. M. 1991. An introduction to randomized algorithms. *Discrete Applied Mathematics* 34, 1-3 (Nov.), 165–201.
- KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. 2000. The Click modular router. *ACM Transactions on Computer Systems* 18, 3 (Aug.), 263–297.
- KOKKU, R., RICHE, T., KUNZE, A., MUDIGONDA, J., JASON, J., AND VIN, H. 2003. A case for run-time adaptation in packet processing systems. In *Proc. of the 2nd Workshop on Hot Topics in Networks (HOTNETS-II)*. Cambridge, MA.
- KWOK, Y.-K. AND AHMAD, I. 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys* 31, 4 (Dec.), 406–471.
- LAKAMRAJU, V., KOREN, I., AND KRISHNA, C. M. 2002. Filtering random networks to synthesize interconnection networks with multiple objectives. *IEEE Trans. Parallel Distributed Systems* 13, 11 (Nov.), 1139–1149.
- MALLOY, B. A., LLOYD, E. L., AND SOUFFA, M. L. 1994. Scheduling DAG's for asynchronous multiprocessor execution. *IEEE Transactions on Parallel and Distributed Systems* 5, 5 (May), 498–508.
- MOTWANI, R. AND RAGHAVAN, P. 1995. *Randomized Algorithms*. Cambridge University Press, New York, NY.
- NILSSON, S. AND KARLSSON, G. 1999. IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications* 17, 6 (June), 1083–1092.
- RAMASWAMY, R., WENG, N., AND WOLF, T. 2004. Application analysis and resource mapping for heterogeneous network processor architectures. In *Proc. of Third Workshop on Network Processors and Applications (NP-3) in conjunction with Tenth International Symposium on High Performance Computer Architecture (HPCA-10)*. Madrid, Spain, 103–119.
- RAMASWAMY, R., WENG, N., AND WOLF, T. 2005. Analysis of network processing workloads. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Austin, TX, 226–235.
- RAMASWAMY, R. AND WOLF, T. 2003. PacketBench: A tool for workload characterization of network processing. In *Proc. of IEEE 6th Annual Workshop on Workload Characterization (WWC-6)*. Austin, TX, 42–50.
- REIJNS, G. L. AND VAN GEMUND, A. J. C. 1999. Analysis of a shared-memory multiprocessor via a novel queuing model. *Journal of Systems Architecture* 45, 14, 1189–1193.
- SHAH, N., PLISHKER, W., AND KEUTZER, K. 2003. 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)*. Anaheim, CA, 100–111.
- TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMAN, H., LEE, J.-W., JOHNSON, P., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., STRUMPEN, V.,
- ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Month 20YY.

- FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2002. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro* 22, 2 (Mar.), 25–35.
- Teja Technologies 2003. *TejaNP Datasheet*. Teja Technologies. <http://www.teja.com>.
- THIELE, L., CHAKRABORTY, S., GRIES, M., AND KÜNZLI, S. 2002. 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)*. Cambridge, MA, 30–41.
- VAN GEMUND, A. J. C. 1993. Performances prediction of parallel processing systems: The pamela methodology. In *Proc. 7th ACM International Conference on Supercomputing*. Tokyo, Japan, 318–327.
- WEI, Y.-C. AND CHENG, C.-K. 1991. Ratio cut partitioning for hierarchical designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10, 7 (July), 911–921.
- WOLF, T. AND FRANKLIN, M. A. 2000. CommBench – a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Austin, TX, 154–162.
- WOLF, T., WENG, N., AND TAI, C.-H. 2005. Design considerations for network processor operating systems. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*. Princeton, NJ, 71–80.

Received August 2005; May 2006; accepted August 2006