Runtime Support for Multicore Packet Processing Systems

Tilman Wolf, University of Massachusetts
Ning Weng, Southern Illinois University
Chia-Hui Tai, Stanford University

Abstract

Network processors promise a flexible, programmable packet processing infrastructure for network systems. To make full use of the capabilities of network processors, it is imperative to provide the ability to dynamically adapt to changing traffic patterns in the form of a network processor runtime system. The differences from existing operating systems and the main challenges lie in the multiprocessor nature of NPs, their on-chip resource constraints, and real-time processing requirements. In this article we explore the key design trade-offs that need to be considered when designing a network processor operating system. In particular, we explore the performance impact of application analysis on partitioning, traffic characterization, workload mapping, and runtime adaptation. We present and discuss qualitative and quantitative results in the context of a particular application analysis and mapping framework. The observations and conclusions are generally applicable to any runtime environment for network processors.

The success of the Internet as a communication medium is driving research in the areas of sensor networks, overlay networks, ubiquitous computing, grid computing, and storage area networks. This trend expands the functionality of networks to include increasingly diverse and heterogeneous end systems, protocols, and services. Even in today's Internet, routers perform a large amount of processing in the *data path*. Examples are firewalling, network address translation (NAT), Web switching, IP traceback, TCP/IP offloading for high-performance storage servers, and encryption for virtual private networks (VPNs). Many of these functions are performed in access and edge networks, which exhibit the most diversity of systems and required network functions. With the broadening scope of networking it can be expected that this trend will continue, and more complex processing of packets *inside* the network will become necessary.

The processing infrastructure for these various packet processing tasks can be implemented in a number of ways. Well defined high-speed tasks are often implemented on application-specific integrated circuits (ASICs). Tasks that are not well defined or possibly change over time need to be implemented on a more flexible platform that supports the ability to be reprogrammed. Network processors (NPs) have been developed for this purpose.

The performance demands of increasing link speeds and the need for flexibility require that these NPs are implemented as multiprocessor systems. This makes the programming of such devices difficult, as the overall performance depends on the fine-tuned interaction of different system components (processors, memory interfaces, shared data structures, etc.). The main problem is handling the complexity of various interacting NP system components. To achieve the necessary processing performance to support multigigabit links, NPs are implemented as system-on-a-chip multiprocessors. This involves multiple multithreaded processing engines, different types of on- and off-chip memory, and a number of special-purpose co-processors.

On conventional workstation or server systems these complexities are hidden by the operating system or do not express themselves as drastically due to their uniprocessor architecture. To simplify this process, a number of domain-specific programming languages and optimizing compilers are currently being developed. These approaches aim at optimizing a single application (i.e., router functionality) statically for the underlying hardware. In current NPs, most performance-critical tasks are implemented and fine-tuned in assembly (e.g., to balance the processing times in each step of a software pipeline). As a result, slight changes in the functionality can have drastic performance impacts that require retuning. Due to the necessary fine-tuning of individual applications, it is very difficult to integrate and dynamically change multiple packet processing functions on a single NP. However, network processing is inherently a dynamic process.

The main motivation for implementing packet processing functions in an NP (rather than in a faster, more power-efficient custom logic device) is the need to change the functionality over time. Changing traffic patterns, new network services and protocols, new algorithms for flow classification, and changing defenses against denial of service attacks present the dynamic background a programmable router needs to accommodate. This requires that the router:

This research was supported in part by National Science Foundation grant CNS-0447873.

- Can implement multiple packet processing applications at the same time
- Can quickly add and remove processing functions from its workload
- Can ensure efficient operation under all circumstances

In particular, the management of various system resources is important to avoid performance degradation from resource bottlenecks.

In this article we explore a variety of design issues for a runtime environment that supports several concurrent network processing applications and allows dynamic reconfiguration of the workload on a multiprocessor system. The key design considerations that are addressed fall into four broad categories:

- Application partitioning
- Traffic characterization
- Runtime mapping and adaptation
- System constraints

The remainder of the article presents some background on related work and differences between NP runtime systems and conventional operating systems. Then qualitative design trade-offs are considered, followed by a discussion on quantitative results from our experimental system. We then summarize our observations and findings.

Background

Related Work

Commercial examples of NPs are numerous (Intel IXP series, EZchip NP-2, Hifn 5NP4G, etc.). An NP is typically implemented as a single-chip multiprocessor with high-performance I/O components, which is optimized for packet processing. In particular, NPs provide a more suitable architecture to handle these workloads than conventional workstation or server processors. The need for a specialized architecture is due to the uniqueness of the workload of NPs, which is dominated by many small tasks and high-bandwidth I/O operations.

In order to achieve the necessary performance of ever-increasing line speeds and increasingly complex packet processing functions, NPs exploit the parallelism that is inherent in the workload of these systems. In general, packets can be processed in parallel as long as they do not belong to the same flow. Processing functions *within* a packet can also be parallelized to decrease packet delay. This leads to NP systems with numerous parallel processing and co-processing engines. To program such a system, several domain-specific programming languages have been developed. Intel jointly with the Shangri-La project at the University of Texas at Austin has developed Baker [1]. The MESCAL project at the University of California at Berkeley has developed NPClick [2], which is based on the Click modular router [3].

Teja is a commercial programming environment for the Intel IXP family of network processors. While they provide a thin network processing operating system (NPOS), Teja is designed to simplify the programming process of a single application and aim at code reuse across platforms. The ability to quickly adapt to multiple applications on the same network processor system is not supported.

Even though there has not been much work on the mechanisms for dynamically managing multiple applications on an NP, there has been work on algorithms for adapting and scheduling NPs to save power. Kokku *et al.* have explored runtime environment design issues [4] similar to the ones we present here, but do not considers partitioned applications that are distributed over several processors of the network processing system. Instead, it is assumed that an entire application is mapped to a single processor core.

In the broader context of embedded systems, runtime scheduling has been explored for real-time scheduling. Chakraborty *et al.* have developed a practical approach to determining if a task graph can be meet real-time constraints [5]. While they consider dynamic interactions through events, they do not consider the fully dynamic case of changing workloads under different network traffic as we do in this article. Most existing real-time operating systems (RTOSs) are designed for single-core platforms and thus not applicable to network processors. Scheduling as a hardware/software co-design problem has been shown to outperform RTOS scheduling [6], but introducing hardware components into NPs for runtime support is not likely to happen in the near future.

Network Processor Operating System

The term *operating system* (which we use synonymously with *runtime system*) is most commonly used in the context of workstation computers. The responsibilities of such an operating system are to manage hardware resources and isolate users from each other. The optimization target is commonly to minimize the execution time of a single task (the one the user is currently working on). It is important to note that the goals of an operating system for NPs are very different. On an NP, all applications are controlled by the same administrative entity, and optimization aims at maximizing overall system throughput.

The following list details the differences between NPOSs and conventional operating systems:

Separation between Control and Data Path. This separation refers to the processor context, not the networking context. To achieve high throughput on NPs, several studies have shown that it is more economical to implement a larger number of simpler processing engines than fewer more powerful ones. Such simple processors do not have the capability to run complex control tasks on top of packet processing tasks. In today's NP designs, control is implemented on a separate control processor. Due to this separation between *classes* of processors, it is necessary to have a more explicit control structure than one would have in a conventional operating system.

Limited Interactivity. Users do not directly interact with an NP or its operating system. At most, applications are installed and removed occasionally. This does not mean that a user could not change configurations on an NP (e.g., update rules for a firewall application), but the key variable in this system are the traffic patterns that determine what processing needs to happen.

Regularity and Simplicity of Applications. One dominating aspect of network processing is that data path processing is performed on individual packets. This means that packet processing tasks are typically limited in complexity due to the real-time constraints imposed by continuously arriving packets. As a result, the processing demands are low (a few hundred to several thousand instructions [7]). Additionally, the execution path within an application is the same in a large number of cases and only slightly different for the other cases. Therefore, it is feasible to analyze packet processing applications in detail to find good processor mappings.

Processing Dominates Resource Management. Conventional operating systems need to implement a number of different functions: processor scheduling, memory management, application isolation, abstraction of hardware resources, and so on. In network processor systems, these challenges are dominated by managing processing resources. The diversity of hardware resources is limited, and many are controlled directly by the application. Also, memory is usually allocated statically to ensure deterministic runtime behavior. This might change in the future as network applications become more complex and

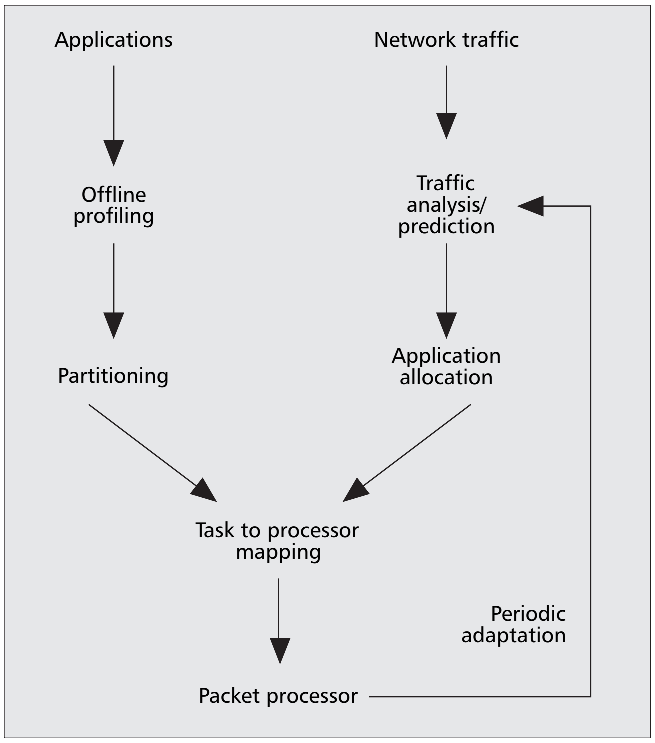


Figure 1. *Processing and traffic analysis in an NP runtime system.*

NPOSs become more similar to conventional operating systems. In this work we focus on processing aspects of operating system functionality.

Nonexistence of User Space/Kernel Space Separation. All functions on an NP are controlled by the same administrative entity. There is no clear separation between user space and kernel space in the traditional sense. Instead, functionality is divided between control and data path. As a result, traditional protection mechanisms are typically not implemented in NPOSs. Due to these numerous and significant differences between what is conventionally thought of as an operating system and what is necessary for an NP, we believe it is important to explore some of the fundamental design issues encountered in the context of NPOSs.

Qualitative Trade-offs

System Operation

In order to explore runtime system design aspects concretely, we assume a general operational approach as shown in Fig. 1. There are four basic steps that are necessary for runtime support of NP systems: application analysis, traffic characterization, workload mapping, and adaptation. There is a fundamental question of what should be done offline (e.g., during application development) and what should (and can realistically) be done during runtime. We discuss the different design choices for these components and then provide quantitative results later. Since the quantitative results are highly dependent on a particular system, we have separated the discussion of trade-offs to preserve its general applicability.

Application Partitioning

Application analysis is necessary to analyze the processing requirements of the application and be able to partition the application. Partitioning allows the distribution of different subtasks onto different processing elements to fully utilize the resources on the NP. The simplicity and repetitiveness of network processing applications allows detailed analysis of the application. The profiling process is shown as an offline component. With the limited processing resources on current NP architectures, this analysis cannot be done online. Typically, such an analysis can be performed in the context of the application development environment for the NP applications. The partitioning can be performed in different ways, and the level of granularity at which it should be performed is discussed in more detail below.

Network processor applications rarely consist of a single monolithic piece of code. More commonly, NP applications are split into several subtasks. For example, on the Intel IXP2400, input processing is separated from forwarding and output processing. This partitioning makes application development somewhat easier and changes to the application easier to implement. Also, it allows exploiting of parallelism and pipelining to fully utilize the multiprocessor infrastructure. How can a runtime system support this application partitioning?

Manual Partitioning: Manual application partitioning is the most common approach to determining a suitable separation of tasks and a mapping of tasks to processors. Using simulation environments, programmers can implement a certain partitioning and obtain performance results. By manually adapting the partitioning, the bottlenecks can be removed and the performance fine-tuned. This approach is very time consuming, and requires a detailed understanding of the application and the NP hardware. From a runtime system perspective, manually partitioned applications limit the amount of dynamic support. It is generally not possible to adapt to changing traffic conditions.

Automated Partitioning: More recently, several approaches to automated partitioning of applications have been investigated. The automatic mapping aspect discussed below is related to this. An auto-partitioning compiler has been developed by Intel [1]. Ramaswamy *et al.* have developed a profiling-based instruction clustering algorithm that derives a directed acyclic graph representation of NP applications [7]. The granularity of the partitioning can be adapted as needed. Plishker *et al.* take an approach where applications are described in a domain-specific language and then distributed onto processing elements [8].

Design Choices: One key question is whicht granularity of application partitioning is most suitable. The spectrum of choices ranges from monolithic applications to extremely fine-grained instruction (or basic block) allocations to processing resources.

If the application is not partitioned, it can only be allocated to a single processing engine. This significantly limits how the application workload can be adapted to network traffic requirements. Also, it may cause performance bottlenecks in pipelined systems (e.g., multiple sequential applications per packet), where the pipeline speed is determined by the maximum stage time. Finally, as application size continues to grow, monolithic applications do not allow for scalable distribution of processing and may conflict with instruction store limitations.

The extreme opposite case is a partitioning where each instruction is mapped individually to a processing resource. This approach provides more flexibility, but also generates more overhead for managing the installation and control of the application. It also increases the complexity of the mapping problem, because a large number of nodes have to be mapped and the space of possible solutions grows significantly with the number of mapping choices. Ideally, we would like to find a balanced partitioning that allows efficient distribution of processing tasks, but keeps the complexity of the mapping problem at bay.

Traffic Characterization

Network traffic characterization is another important aspect of runtime support for NP systems. Depending on the requirements of current network traffic, different applications dominate the processing. The dynamically changing workload is the main reason runtime support is necessary. In order to achieve a good allocation of processing resources, it is necessary to know what processing is necessary for packets currently in the system (or to be processed in the near future). The result of traffic analysis is an application allocation, which describes the ratio of processing required by each application available on the system.

Traffic characterization is an important input to determining a suitable allocation of different applications on the network processor. Heavily used applications typically need to be replicated multiple times to provide sufficient performance (e.g., multiple parallel IP forwarding applications). When considering runtime support for workloads, it is important to be able to analyze network traffic to estimate and possibly predict a suitable application allocation.

Static Traffic Model: The simplest case of traffic characterization is a static traffic model. This is the most commonly used model since it does not require any online changes in the system. The assumption is that traffic requirements remain the same over the entire runtime of the system. Short-term variations are compensated by buffering (thus increasing the packet delay) or overprovisioning, where additional resources are allocated to each type of application (thus increasing the total required hardware resources).

Dynamic Traffic Model: In many cases, network traffic exhibits a certain amount of variation and temporal locality. In a dynamic traffic characterization, requirements are specified for a limited number of packets (i.e., *batch*). These requirements can change with each set of packets. The processing requirements for each batch can be determined either deterministically by buffering and analyzing all packets in the batch or statistically by sampling the processing requirements of a small subset of packets and extrapolating accordingly.

Design Choices: In order to determine the allocation of applications to the NP system, traffic characterization can be performed according to the approaches described above. While static traffic models are simplest, they do not serve situations where any changes in workload occur. When considering dynamic traffic models, it is important to consider the trade-off between accuracy and delay. The more packets can be buffered and analyzed for determining workload requirements, the more accurately such processing needs can be determined. This, however, comes at the cost of increasing delay as system adaptation is delayed until an accurate processing estimate is available.

Runtime Mapping and Adaptation

Workload mapping is the process that assigns processing tasks to actual processing engines. This assignment is based on the application allocation and application partitioning derived in the previous two steps. Mapping can be performed in a number of different ways and depends on the particular system architecture, application development environment, and operational principles of a system. The goals of mapping are to achieve high system throughput and efficient resource utilization.

The adaptation step illustrates the need to reconfigure the NP system to match the processing requirements dictated by the traffic workload. During adaptation, the application allocation is changed according to the new traffic requirements. Then the mapping step modifies the allocation of tasks to processors to match the new allocation.

The mapping of application tasks to processing elements is performed through a mapping algorithm. We explore the design trade-offs without requiring that a particular algorithm be used. However, we assume two properties of the mapping algorithm:

- The mapping algorithm can yield incrementally better results as the runtime increases. This implies that the runtime system designer could choose the runtime of the algorithm and the quality of the resulting mapping.
- The mapping algorithm can be employed on a partially configured system. This means that some applications can be removed and others added without changing the mapping of applications that are not affected.

The resulting design choices address the frequency and level of (partial) mapping.

Static Mapping: Static mapping goes hand in hand with static partitioning and a static traffic model. In this case processing tasks are allocated to processors offline, and no changes are made during runtime.

Complete Dynamic Mapping: Complete mapping refers to a mapping solution where the entire workload is mapped from scratch. The mapping algorithm can place processing tasks on the entire architecture without any initial constraint. This typically leads to a good solution that approximates the theoretical optimum for increasing processing times.

Partial Dynamic Mapping: Partial mapping assumes that some part of the workload is already mapped to the NP system. The mapping algorithm only needs to map a few applications to the remaining processing resources. This approach is more restrictive than complete dynamic mapping because many of the system resources are already assigned to applications. The incremental nature of this approach poses the risk that the mapping algorithm "gets stuck" in a local minimum. Nevertheless, the processing cost for the partial mapping is less than mapping the entire workload.

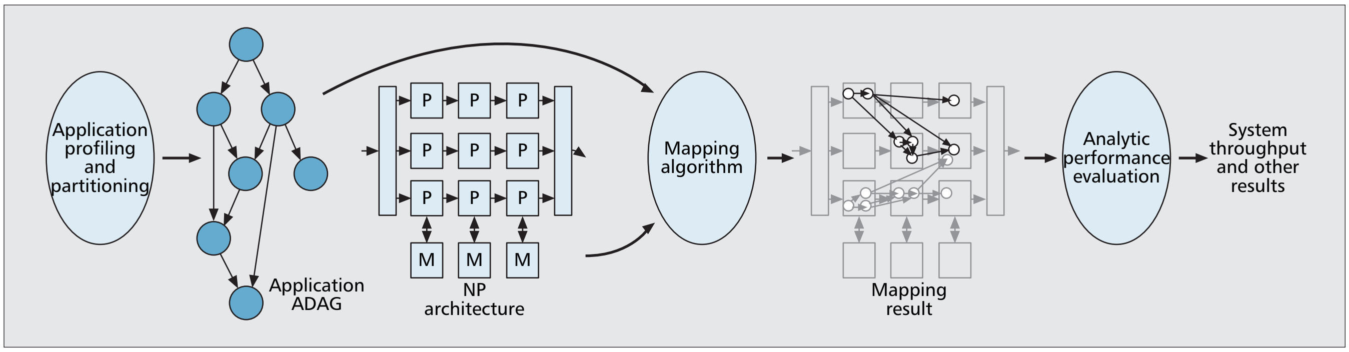
Design Choices: Design choices for mapping determine how often, how much, and with how much effort to perform complete or partial mapping.

In order to adapt to changing traffic conditions, the NP runtime system needs to change the application allocation and thus the mapping. Ideally, we want to reconfigure application allocation with every packet to guarantee the best system utilization and high performance while traffic is varying. However, there is a cost associated with mapping and remapping. Apart from the cost of uploading new instructions to each processor, determining the new mapping requires processing power and computation time. It is important to keep the reprogramming frequency at a low enough rate that sufficient processing time is available to find good mapping results. The lower the adaptation rate, the more time can be spent on finding a better mapping solution. As the adaptation rate increases, the quality of the derived mapping solution decreases.

Changes in traffic conditions may only affect a few applications. In order to be able to adapt quickly with low mapping cost, a runtime system designer may choose to only map a small part of the overall allocation. The benefit of this is the ability to adapt quickly, but the amount of traffic variation that can be supported is limited to the fraction of the NP that is remapped. Repeated partial mapping causes mapping solutions to deteriorate. In order to avoid this, complete mapping steps should be performed periodically. The more frequently this happens, the less likely the system will go into an inefficient state. However, this also increases the overall mapping effort.

Constraints

An NP has a number of system constraints that are not considered in the above discussion. These constraints can play a major role when making design decisions.


Application profiling and partitioning
Application ADAG
P
M
NP architecture
Mapping algorithm
Mapping result
Analytic performance evaluation
System throughput and other results

■ Figure 2. *Application analysis, mapping, and performance evaluation process. Typically, multiple ADAGs are mapped to the NP architecture to reflect the workload mix that can be processed by the network processor.*

Most NP systems are severely limited in the amount of instruction storage available for each processing engine. This is due to the relatively high chip area cost of memory compared to processing logic. This limitation is the reason not all applications can be installed on all processing engines at all times. Therefore, mapping changes are quite costly as they require uploading of new instructions to every processor.

At the same time, an NP system needs to be capable of processing any packet transmitted on the network. In the context of an NP runtime system, this means that processing resources should be available for all applications. If this is not the case, packets need to be delayed until the next adaptation cycle or processed in the slow path of the router. One way to avoid this delay is to overprovision the system and increase the application allocation to more than 100 percent. It can be ensured that even applications not expected to see any traffic in the upcoming batch can be installed just in case.

Quantitative Results

In this section we support the qualitative observations of the previous sections with quantitative results. This helps illustrate which trends have a large impact and which have a small impact on system performance.

In order to derive quantitative results, we use a particular system baseline. Of course, there are big differences between different NP systems, and results on another system would look somewhat different. It is therefore more important to consider the trends that can be observed in our results (e.g., does an optimum exist?) than individual data points (e.g., where exactly is the optimum?).

Baseline System

The metric in which we are interested is the throughput of an NP system given a certain workload. In order to derive this information, we need to implement some of the functions necessary for network runtime systems. In particular, we need to consider realistic network processing applications, their partitioning, and the mapping of processing tasks to processing engines. In order to explore the design space we have described, it is not sufficient to consider only a handful of partitioning and mapping solutions. Therefore, we choose to use an analytic modeling approach instead of simulation. With the automated partitioning, mapping, and performance modeling environment provided in [9], we can evaluate a large number of possible application partitionings, mapping results, and so on. This provides a first-order understanding of the quantitative trade-offs. In the process, several ancillary metrics (e.g., cost for deriving a certain quality mapping) can be obtained.

The process of obtaining performance results is shown in Fig. 2. We briefly describe the three key components, application partitioning and representation, the mapping algorithms, and the analytic performance model, to provide a basis for understanding the results below.

Application Representation: A network processing application needs to be represented in such a way that it can easily be mapped to multiple parallel or pipelined processing elements. This requires a representation that exhibits 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.

The ADAG is derived from dynamic profiling of the application by determining data and control dependencies between individual instructions. The regularity and simplicity of network processing applications allows for loop unrolling and efficient ADAG representation. Using a clustering heuristic that minimizes the communication overhead, instructions are aggregated to larger nodes in the graph. Each node is annotated with information on the total number of instructions and memory accesses that need to be executed when processing the node.

Mapping Algorithm: Once we have the application represented as an ADAG, the next step is to map the ADAG onto a NP topology. 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. Furthermore, Malloy *et al.* have shown that producing an optimal schedule for a system that includes both execution and communication cost is NP-complete, even if there are only two processing elements [10]. 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. 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 [11]. The mapping is performed for multiple, possibly different, ADAGs. The mixture of ADAGs represents the allocation of applications to the NP architecture.

Analytic Performance Model: In order to evaluate the throughput performance of a given solution, we use an analytic performance model that considers processing, interprocessor communication, memory contention, and pipeline synchronization effects. After mapping the application ADAGs to the network processor topology, we know exact-

System parameter	Baseline configuration
NP pipeline stages	4
PEs per stage	4
Total number of PEs	16
Memory interfaces per stage	2
Memory access time (in cycles)	10
Number of application ADAGs	20
Nodes per application ADAG	8

Table 1. *Baseline configuration for quantitative results.*

ly the workload for each processing element. This information includes the total number of instructions executed, the number of memory accesses, and the amount of communication between stages. The model needs to take this into account as well as contention for shared resources (memory channels and communication interconnects). We are particularly interested in the maximum latency of each pipelined processing stage since that determines the overall system speed. The number of ADAGs that are mapped to an architecture determines how many packets are processed during any given stage time. After specifying the several system parameters, the throughput of the system for a given mapping can be expressed. The details of this analytic model and its validation against cycle accurate simulation can be found in [9].

System Configuration and Workload: The system architecture considered in the above model can be configured to represent any regular NP architecture with a specified number of parallel processing engines (PEs) and pipeline stages. We have chosen one single baseline system with a fixed configuration to explore the runtime system issues (Table 1). A separate question is how these change for different architecture configurations. This design space exploration is currently not addressed in our work. The applications that are considered for this system are radix-tree-based IP-lookup and hash-based flow classification.

Processing Task Mapping

Metrics: Mapping takes a certain amount of processing time, which can be seen as the cost of mapping. The performance achieved by the mapping is expressed as the throughput of the system. Due to the NP-completeness of the mapping problem, finding the overall optimal solution is infeasible. What is really desirable in a system is to obtain a *good enough* solution by running the approximation algorithm for a large amount of time.

Results: Figure 3 shows the increasing quality of the mapping result as more processing effort is dedicated to the mapping process. The x-axis shows the time it takes to calculate a mapping (expressed as the number of randomized mapping iterations). We quantify this cost on the Intel IXP2400 and also consider the overhead for stopping, reprogramming, and restarting processing engines. The y-axis shows the best throughput that was found within a given number of mapping attempts. This is expressed in relation to the maximum system performance, which is derived by using a very large number of mapping attempts. With increasing mapping cost, more randomized mappings can be attempted, and mappings with higher throughput can be found.

Partitioning

The goal of application partitioning is to study the impact of partition granularity on the performance.

Metrics: We consider the number of tasks (or nodes in the ADAG), n, into which the application is partitioned. The maximum n is different for each application, but for this evaluation we only consider values of n that are much smaller than this maximum. If the partitioning is balanced, the size of each subtask is approximately $1/n$ of the application size.

Results: First, we explore the trade-off between system throughput and partitioning granularity. Finer granularity promises better performance if the permissible mapping effort is unbounded. When considering the realities of a network processor runtime system, the mapping effort is bounded by the batch size and adaptation frequency.

Figure 4 shows the throughput of the baseline system with different levels of application granularity relative to a monolithic implementation, where the entire application is executed on a single processor. The mapping effort is fixed, and it can be observed that the best performance is achieved for $n = 5$. The monolithic application performs worse because it does not permit an even distribution of processing tasks on the multiprocessor system. Larger values of n also degrade the performance because the mapping problem becomes more complex, and finding a good mapping in a limited amount of time becomes more difficult.

Traffic Characterization

The need for runtime adaptation is determined by the characteristics of network traffic.

Metrics: We assume that traffic is processed in batches (with batch size b), and the application allocation is based on a sample (size l). We can then describe the traffic variation v based on two metrics, $e_{i,j}(a)$ and $p_{i,j}(a)$. Metric e reflects the estimated number of packets requiring application a, and metric p is the actual number of packets requiring this application in packet interval $[i \ldots j)$:

$$v_i(l,b) = \frac{1}{b} \cdot \sum_a \max(p_{i,i+b}(a) - \frac{b}{l} e_{i,i+l(a)}, 0). \tag{1}$$

For example, if the traffic exactly matches the estimated allocation, all packets "match up" and the traffic variation is $v = 0$. If half the packets of a batch are different from what was expected (e.g., all packets require a single application instead

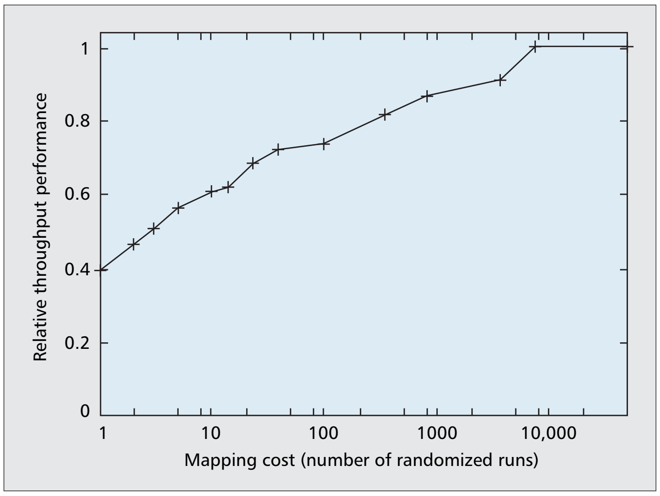
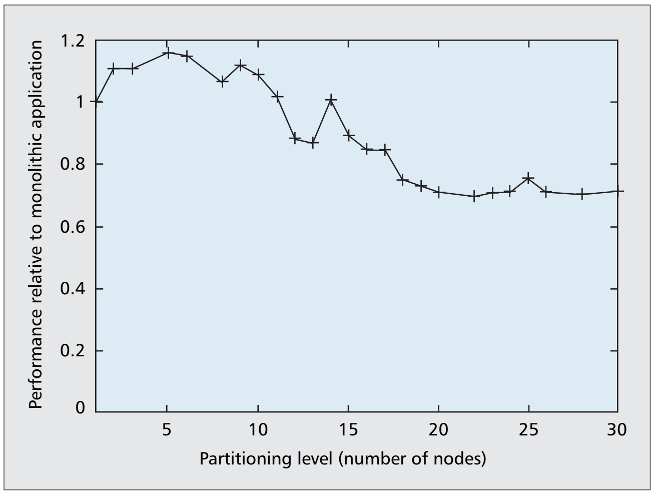


Figure 3. *System performance compared to mapping cost.*



■ Figure 4. *Performance for different levels of partitioning of a single application. The overall mapping effort is limited to 10,000 total node mapping attempts.*

of an estimated 50/50 split between two applications), the traffic variation is $v = 0.5$.

Results: To illustrate traffic variation of realistic network data, we have obtained measurement data from the main Internet access link of the University of Massachusetts. We collected 4,235,403 packets and classified them by layer 7 applications (using the classification rules of the Ethereal tool). There are a total of 175 categories, but over 98 percent of the traffic falls into the top five categories. Figure 5 shows traffic variation over a sequence of packets. The parameters are $b = 10{,}000$ and $l = 100$, and the variation is computed as a sliding window. While this is only a small sample of the overall measurement we have performed, it does reflect the overall trends correctly. In most cases, the variation is around $v = 0.04$ with a few spikes of up to $v = 0.15$.

Of course, the observed variation depends on the quality of the estimate (i.e., size of l relative to b) and batch size b. With small samples and small batch sizes, we have observed that the average variation is very high (around $v = 0.4$). The peak variation in this case can reach $v = 1$. The larger the sample percentage, the better the estimates and thus the lower the traffic variation. As the batch size increases, the temporary variations within the network traffic are "smoothed out" and less average variation is observed. This does not mean that the static allocation approach ($b = 1$) is necessarily ideal. In the static case, deviations from the allocation can cause large delays, and overprovisioning is necessary to support all possible traffic conditions.

Adaptation

Adaptation during runtime is guided by the variation of traffic. Complete and partial mapping is performed during each adaptation process.

Metrics: The metrics that are interesting for adaptation are the system throughput in comparison to a baseline configuration. The key input parameters are the frequency of adaptation and the amount of partial mapping (i.e., the fraction of NP resources to which new applications can be allocated). For our experiment, we consider the adaptation frequency to be the same as the batch size.

Results: Figure 6 shows the degradation effect of repeated partial mapping. The figure shows that repeated removals and additions of applications cause system performance to quickly drop and then stabilize at a suboptimal state of about 80 percent of peak performance. There is little variation between different levels of partial adaptation; the stabilizing value is driven more by the amount of effort dedicated to the partial mapping. This result clearly shows that partial mapping can very quickly lead to suboptimal configurations.

When designing an NP runtime system, it should be taken into account that complete mapping is occasionally necessary. The adaptation frequency (i.e., batch size) poses a trade-off between the adaptiveness of the system to changing traffic and the quality of the mapping result that can be derived. Figure 7 shows the relative performance of configurations with different batch sizes and traffic variation (as defined in Eq. 1). With increasing batch size, the performance of the system increases, but as the traffic variation increases, the performance decreases.

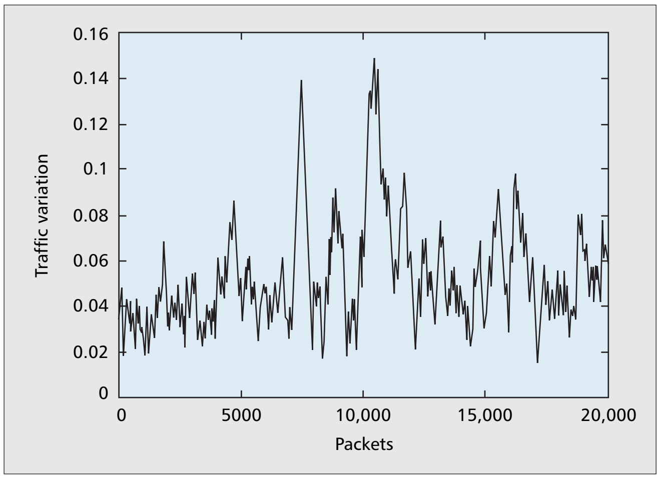
For small batch sizes, processing requirements can be estimated correctly and thus variation is low. But even for a perfect estimation ($v = 0$), the performance is suboptimal. This is due to packet being processed as discrete entities and a small number of application allocations may not fully utilize the network processor. As the batch size increases, this effect decreases and the performance approaches that of an infinite batch size (with perfect estimation). In practice, however, variation increases due to longer-term changes in traffic workload and less accurate prediction. The optimal performance in a practical system lies between the two extremes where low traffic variation and large batch sizes coincide. This can be observed for *operation variation*, which corresponds to the variation obtained from the measurement data shown in Fig. 5.

Run-Time System Design Scenarios for Intel IXP2400 Network Processor

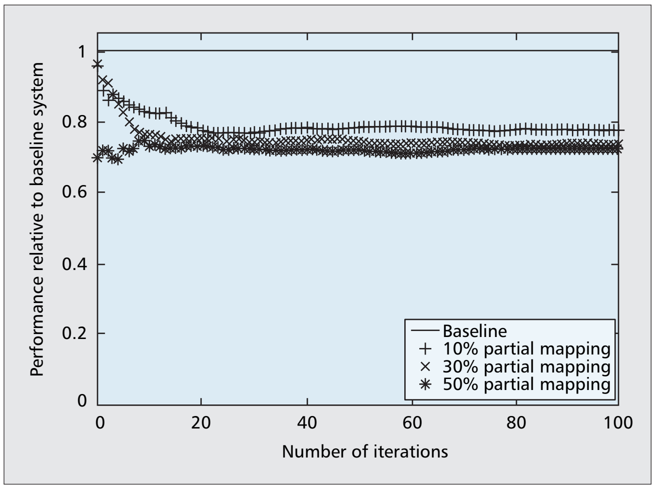
In the previous two sections, we have discussed a number of design considerations and explored their qualitative and quantitative dependencies. To summarize some of these observations, we present three specific configurations for run-time systems in the context of the Intel IXP2400 network processor, an NP widely used in industry and academia. The processing resources on this system consist of 16 processing engines for data path operations and one control processor. We assume the run-time system processing (e.g., mapping algorithm) to be executed on this control processor.

Scenario 1: Static Configuration

This scenario assumes that all analysis, mapping, and allocation operations are performed offline. Once the network pro-



■ Figure 5. *Traffic variation over a sequence of packets.*



■ Figure 6. *Performance degradation due to repeated partial mapping. The baseline case is a complete mapping.*

cessor is configured, no adaptation is performed. The design and performance considerations are:

- Simplicity of system: Clearly, as static handling of all mapping and allocation issues simplifies the implementation of the system, there is no need for runtime control.
- Limited runtime flexibility: The static approach does not allow for any changes during runtime. Any change in the workload configuration requires the use of a software development tool to reconfigure the entire system. As NPs become more integral components of networks, this approach will become less feasible.
- Performance degradation under traffic variation: Where traffic requirements change over time, performance degradation can be observed unless workloads are allocated with significant overprovisioning.

This scenario represents today's state of the art when developing applications for the IXP2400 using the provided software development kit.

Scenario 2: Predetermined Configurations

In this scenario, we assume that the NP system can be configured to one of multiple predetermined workload configurations. Each configuration is statically mapped in an offline process. During runtime, the system can adapt to any one of these configurations. The design and performance considerations are:

- Offline mapping: While the system needs to monitor traffic variation, it does not need to perform mapping computations. This limits the complexity of the runtime system.
- Limited adaptability: The number of configurations that can be precomputed is limited by the available memory on the control processor. The IXP2400 is equipped with an instruction store of 4000 × 40 bits (=20 kbytes), which needs to be stored for each preconfigured runtime setup. If traffic varies outside the estimated bounds, no further adaptation is possible. Within the bounds of estimated traffic, it is unlikely that actual traffic completely matches a predetermined configuration. Thus, a certain level of performance degradation can be expected.
- Better quality mapping results: Due to the availability of arbitrary amounts of computational power for offline mapping, the quality of mapping results (Fig. 3) can be better than for online mapping.

On the Intel IXP2400, this adaptation process requires the exchange of program data in the instruction memory of the processing engines. This process has been reported to be possible on Intel NPs with as little overhead as 30 μs [13]. If the configuration of this runtime system on the IXP2400 allowed for 1 percent downtime for reconfiguration, the system could be reconfigured 330 times/s. If 16 Mbytes of DRAM memory was dedicated for preconfigured mappings, a total of 4096 different workload configurations could be maintained by the system. While this is sufficient for most scenarios that involve a small number of different applications, it is not possible to support a completely dynamic workload.

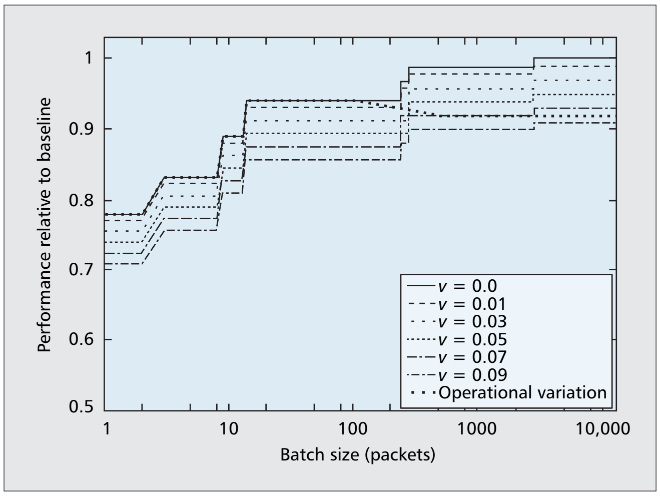
Scenario 3: Fully Dynamic Configuration

In the fully dynamic scenario, mapping is performed online, and the system adaptation is performed to match the traffic variation. The design and performance considerations are:

- Complete adaptability: A fully dynamic system can adapt to any traffic variations — even configurations that could not be predicted when programming the system. This is clearly the most important functional benefit of this scenario.
- Limited mapping quality: Due to the online nature of the mapping process, only limited amounts of processing time are available. Thus, the quality of the mapping results is lower than that of the other two scenarios.
- Lower overprovisioning overhead: Due to the ability to adapt to changing traffic conditions, a fully dynamic scenario can provide high throughput performance with less processing resources. It is not required to overprovision or store preconfigured mappings.

On the XScale control processor of the IXP2400 (clocked at 600 MHz), the mapping algorithm described above requires on average 5.3 ms/mapping run for a single ADAG. According to the results from Fig. 3, this yields a 60 percent quality mapping (for the baseline case in Table 1) for a processing time of 1 s. This processing time is too high to adapt quickly to changing traffic conditions; thus, utilizing partial mapping is suggested.

The three runtime configurations for the IXP2400 are summarized in Table 2. Overall, a hybrid approach between some common predetermined configurations and the ability to dynamically adapt to entirely new traffic requirements balances the capabilities of the Intel IXP2400 best. One possible way of implementing such dynamic adaptation is to cache the most recent mappings, as it can be expected that recently observed traffic patterns may occur again in the near future.



■ Figure 7. *Performance for different batch sizes under traffic variation. The total mapping effort is fixed, and the baseline case has infinite batch size with no traffic variation.*

	Static configuration	Predetermined configuration	Fully dynamic configuration
# of supported runtime configurations	1	4096	Arbitrary
Mapping computation	Offline (approx. 60 s)	Offline (approx. 60 h)	Online (5.3 ms/run)
Maximum adaptation rate	N/A	330 adapt./s (100% quality)	1 adaptation/s (60% quality)
Memory	20 kbytes	16 Mbytes	20 kbytes

Table 2. *Configuration scenarios for different runtime systems on the Intel IXP2400.*

Conclusion

We have presented an extensive qualitative discussion of design issues related to runtime system design for network processors. To illustrate the design considerations, we have provided quantitative results that highlight performance trade-offs between various design parameters. Finally, we have explored three different runtime system designs in the context of the Intel IXP2400, and discussed their benefits and drawbacks. We believe that this study provides an important basis for design and implementation of future runtime systems for network processors. Understanding the presented trade-offs will guide runtime system designers in considering the relevant interactions between applications, network traffic, and the underlying hardware. This will bring us closer to realizing network processors as easy-to-use components of network systems.

References

[1] S. D. Goglin *et al.*, "Advanced Software Framework, Tools, and Languages for the IXP Family," *Intel Tech. J.*, vol. 7, no. 4, Nov. 2003, pp. 64–76.

[2] N. Shah, W. Plishker, and K. Keutzer, "NP-Click: A Programming Model for the Intel IXP1200," *Proc. 2nd Network Processor Wksp.* in conjunction with *9th IEEE Int'l. Symp. High Perf. Comp. Architecture*, Anaheim, CA, Feb. 2003, pp. 100–11.

[3] E. Kohler *et al.*, "The Click Modular Router," *ACM Trans. Comp. Sys.*, vol. 18, no. 3, Aug. 2000, pp. 263–97.

[4] R. Kokku *et al.*, "A Case for Run-Time Adaptation in Packet Processing Systems," *Proc. 2nd Wksp. Hot Topics in Networks*, Cambridge, MA, Nov. 2003.

[5] S. Chakraborty *et al.*, "Schedulability of Event-Driven Code Blocks in Real-Time Embedded Systems," *DAC '02: Proc. 39th Conf. Design Automation*, June 2002, pp. 616–21.

[6] V. J. Mooney, III and G. De Micheli, "Hardware/Software Co-Design of Run-Time Schedulers for Real-Time Systems," *Design Automation for Embedded Sys.*, vol. 6, no. 1, Sept. 2000, pp. 89–144.

[7] R. Ramaswamy, N. Weng, and T. Wolf, "Application Analysis and Resource Mapping for Heterogeneous Network Processor Architectures," *Proc. 3rd Wksp. Network Processors and Apps.* in conjunction with *10th IEEE Int'l. Symp. High Perf. Comp. Architecture*, Madrid, Spain, Feb. 2004, pp. 103–19.

[8] W. Plishker *et al.*, "Automated Task Allocation for Network Processors," *Proc. Network System Design Conf.*, Oct. 2004, pp. 235–45.

[9] N. Weng and T. Wolf, "Analytic Modeling of Network Processors for Parallel Workload Mapping," to appear, *ACM Trans. Embedded Comp. Sys.*

[10] B. A. Malloy, E. L. Lloyd, and M. L. Souffa, "Scheduling DAG's for Asynchronous Multiprocessor Execution," *IEEE Trans. Parallel and Distrib. Sys.*, vol. 5, no. 5, May 1994, pp. 498–508.

[11] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge Univ. Press, 1995.

[12] T. Wolf, N. Weng, and C.-H. Tai, "Design Considerations for Network Processor Operating Systems," *Proc. ACM/IEEE Symp. Architectures for Networking and Commun. Sys.*, Princeton, NJ, Oct. 2005, pp. 71–80.

[13] A. Gavrilovska, S. Kumar, and K. Schwan, "The Execution of Event-Action Rules on Programmable Network Processors," *Proc. 1st Wksp. Op. Sys. and Architectural Support for the On-Demand IT Infrastructure* in conjunction with *ASPLOS-XI*, Boston, MA, Oct. 2004.

Biographies

TILMAN WOLF (wolf@ecs.umass.edu) is an assistant professor in the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst. He received a D.Sc. in computer science in 2002 from Washington University in St. Louis. His research interests are in the areas of computer networks, computer architecture, and embedded systems.

NING WENG (nweng@siu.edu) received an M.S. degree in electrical and computer engineering from the University of Central Florida in 2000. He received a Ph.D. degree in electrical and computer engineering from the University of Massachusetts, Amherst in 2005. He is currently an assistant professor in the Department of Electrical and Computer Engineering at Southern Illinois University, Carbondale. His research interests are system integration, network processing system design, and network security.

CHIA-HUI TAI (chtai@stanford.edu) is a doctoral student at Stanford University. She received her B.S. degree at National Taiwan University and her M.S. degree at the University of Massachusetts, Amherst. Her research interests span the areas of Internet congestion control, queuing systems, and packet switch architectures.