

# NP-Click: A Programming Model for the Intel IXP1200

Niraj Shah, William Plishker, Kurt Keutzer  
University of California, Berkeley  
{niraj,plishker,keutzer}@eecs.berkeley.edu

## Abstract

*The architectural diversity and complexity of network processor architectures motivate the need for a more natural abstraction of the underlying hardware. In this paper, we describe a programming model, NP-Click, which makes it possible to write efficient code and improve application performance without having to understand all of the details of the target architecture. Using this programming model, we implement the data plane of an IPv4 router on a particular network processor, the Intel IXP1200, and compare results with a hand-coded implementation. Our results show the IPv4 router written in NP-Click performs within 7% of a hand-coded version of the same application using a realistic packet mix.*

## 1. Introduction

The past five years has witnessed over 30 attempts at programmable solutions for packet processing [1]. With these architectures, network processor designers have employed a large variety of hardware techniques to accelerate packet processing, including parallel processing, special-purpose hardware, heterogeneous memory architectures, on-chip communication mechanisms, and the use of peripherals [2]. However, despite this architectural innovation, relatively little effort has been made to make these architectures easily programmable. In fact, these architectures are very difficult to program [3].

The current practice of programming network processors is to use assembly language or a subset of C. This low-level approach to programming places a large burden on the programmer to understand fine details of the architecture simply to implement a packet processing application, let alone optimize it. We believe the programmer should be able to implement an application using a natural interface such as a domain-specific language. To accomplish this, we need an *abstraction* of the underlying hardware that exposes enough architectural detail to write efficient code for that platform, while hiding less essential architectural complexity. We call this abstraction a *programming model*.

Our goal is to create an abstraction that enables the programmer to realize the full computational power of the underlying hardware. We realize our programming model will initially introduce some implementation inefficiencies versus a hand-coded approach. However, our programming model offers a compelling design flow that produces implementations that are within 10% of the performance of a hand-coded approach (for a realistic packet mix) at a fraction of the design time.

Further, we believe network processors are just one example of a broader trend to search for application-specific solutions with fast time-to-market. This trend is drawing system designers away from the time-consuming and risky process of designing application-specific integrated circuits (ASICs) and toward programming application-specific instruction processors (ASIPs). As system designers increasingly adopt programmable platforms, we believe the *programming model* will be a key aspect to harnessing the power of these new architectures and allowing system designers to make the transition away from ASICs.

This paper describes NP-Click, a programming model for a common network processor, the Intel IXP1200. We illustrate our approach by using NP-Click to implementing an IPv4 packet forwarder.

The remainder of the paper is organized as follows: Section 2 describes some background. Section 3 introduces the notion of a programming model and motivates it. Section 4 describes our programming model for the Intel IXP1200. We report our results in Section 5. Finally, we summarize and comment on future research direction in Sections 6 and 7, respectively.

## 2. Background

In this section, we describe some relevant background to our work. We first give an overview of Click, a domain-specific language and infrastructure for developing networking applications, upon which our programming model is based. Next, we describe the Intel IXP1200, the target architecture for our application.

## 2.1. Click

Click is a domain-specific language designed for describing networking applications [4]. It is based on a set of simple principles tailored for the networking community. Applications in Click are built by composing computational tasks, or elements, which correspond to common networking operations like classification, route table lookup, and header verification. Elements have input and output ports that define communication with other elements. Ports are connected via edges that represent packet flow between elements.

In Click, there are two types of communication between ports: push and pull. Push communication is initiated by the source element and effectively models the arrival packets into the system. Pull communication is initiated by the sink and often models space available in hardware resources for egress packet flow. Click designs are often composed of paths of push elements and paths of pull elements. Push paths and pull paths connect through special elements that have different typed input and output ports. The *Queue* element, for example, has a push input but a pull output, while the *Unqueue* element has a pull input, but a push output.

Figure 4 shows a Click diagram of the application we implemented with our programming model. The boxes represent elements. The small triangles and rectangles within elements represent input and output ports, respectively. Filled ports are push ports, while empty ports are pull ports. The arrows between ports represent packet flow.

Click is implemented on Linux using C++ classes to define elements. Element communication is implemented with virtual function calls to neighboring ele-

ments. To execute a Click description, a task scheduler is synthesized to run all push (pull) paths by firing their sources (sinks), called schedulable elements.

A natural extension of this Click implementation is to multiprocessor architectures that may take advantage of the inherent parallelism in processing packet flows. A multi-threaded version of Click targets a Linux implementation and uses worklists to schedule computation [5]. Two pertinent conclusions can be drawn from this work: First, significant concurrency may be gleaned from Click designs in which the application designer has made no special effort to express it. Since packet streams are generally independent, ingress packets may be processed by separate threads with very little interaction. Second, a Click configuration may easily be altered to express additional concurrency without changing the application's functionality.

## 2.2. Intel IXP1200

The IXP1200 [6] family is one of Intel's recent network processor product lines based on their Internet Exchange Architecture. It has six RISC processors, called microengines, plus a StrongARM processor (see Figure 1). The microengines are geared for data plane processing and have hardware support for four threads that share a program memory. The StrongARM is mostly used to handle control and management plane operations. The memory architecture is divided into several regions: large off-chip SDRAM, faster external SRAM, internal scratchpad, and local register files for each microengine. Each of these areas is under direct control by the user and there is no hardware support for caching data from slower memory into smaller faster memory (except for the small cache accessible

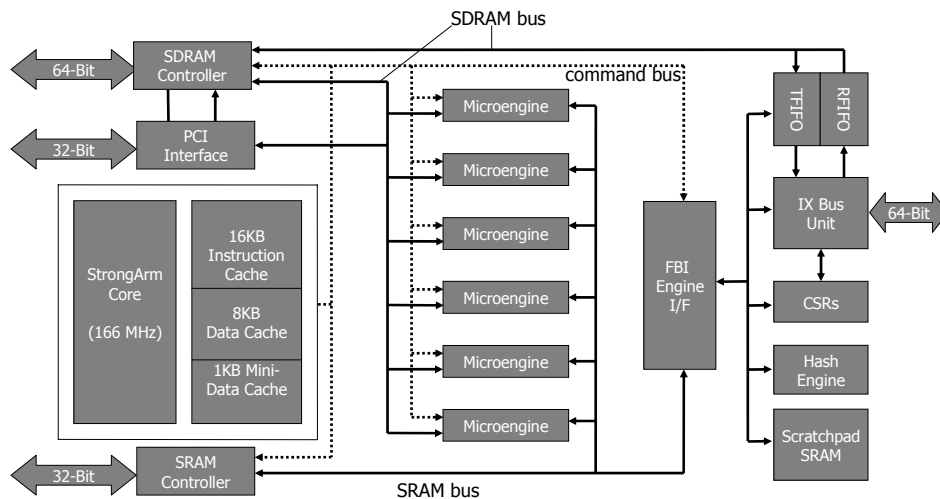


Figure 1. Intel IXP1200 Architecture.

only to the StrongARM). The IX Bus (an Intel proprietary bus) is the main interface for receiving and transmitting data with external devices such as MACs and other IXP1200s. It is 64 bits wide and runs up to 104MHz allowing for a maximum throughput of 6.6Gbps. The microengines can directly interact with the IX bus through an IX Bus Unit, so a thread running on a microengine may receive or transmit data on any port without StrongARM intervention. This interaction is performed via Transmit and Receive FIFOs which are circular buffers that allow data transfers directly to/from SDRAM. For the microengines to interact with peripherals (e.g. determining their state), they need to query or write to control status registers (CSRs). Accessing control status registers requires issuing commands across the command bus which is also used for issuing hash engine, scratchpad memory, and Transmit and Receive FIFOs commands.

Initially, the programming interface provided with the IXP1200 was assembler. This was later augmented with a subset of the C language (which we refer to as IXP-C) [7]. IXP-C supports loops, conditionals, functions, types, and intrinsics (function calls using C syntax that call assembler instructions). However it has several notable deficiencies: it does not support function pointers, it lacks recursion, and it forces the user to control thread swapping. In addition, for practical implementations, the programmer must effectively allocate data to the multiple memory regions (SDRAM, SRAM, scratchpad, data registers, transfer registers), arbitrate access to shared resources by multiple threads, divide code among threads, interact with peripherals, and take advantage of the concurrency inherent in the application. We believe this places undue burden on the programmer to generate even a functional implementation of an application, let alone an efficient one. It is this concern that motivates our own new layer to sit atop IXP-C.

### 3. Programming Models

There is currently a large gap between domain-specific languages that provide programmers a natural interface, like Click, and the complex programmable architectures used for implementation, like Intel's IXP1200. In this section, we introduce and define the concept of a *programming model* to assist in bridging this gap.

#### 3.1. Implementation Gap

We believe Click to be a natural environment for describing packet processing applications. Ideally, we would like to map applications described in Click di-

rectly to the Intel IXP1200. However, there is currently a large gap between Click and the low level programming interface the IXP1200 exposes. The simple yet powerful concept of push and pull communication between elements that communicate only via passing packets, coupled with the rich library of elements of Click provides a natural abstraction that aids designers in creating a functional description of their application. This is in stark contrast to the main concepts required to program the IXP1200. When implementing an application on this device, the programmer must carefully determine how to effectively partition his application across the six microengines, make use of special-purpose hardware, effectively arbitrate shared resources, and communicate with peripherals. We call this mismatch of concerns between the application model target architecture the *implementation gap* (see Figure 2). To facilitate bridging this gap, we propose an intermediate layer, called a *programming model*, which presents a powerful abstraction of the underlying architecture while still providing a natural way of describing applications.

#### 3.2. What is a Programming Model?

A programming model presents an abstraction that exposes only the relevant details of the architecture necessary for a programmer to efficiently implement an application. It is a programmer's view of the architecture that balances opacity and visibility:

1. Opacity: Abstract the underlying architecture  
*This obviates the need for the programmer to learn intricate details of architecture just to begin programming the device.*
2. Visibility: Enable design space exploration of implementations  
*This allows the programmer to improve the efficiency of their implementation by trading off different design parameters (e.g. thread boundaries, data locality, and implementation of elements). Our goal is that the full computational power of a target device should always be realizable through the programming model.*

In summary, a programming model supplies an approach to harvesting the power of the platform. It is a more productive way of harvesting that power. A programming model will inevitably balance between a programmer's two competing needs: desire for ease of programming and the requirement for efficient implementation. Further, we believe the programming model is a necessary, but not sufficient, condition of closing the implementation gap.

### 3.3. Possible Approaches to Solving the Implementation Gap

There are many different approaches to solving the *implementation gap*. We classify prior works in this area into four major areas: library of application components, programming language-based, refinement from formal models of computation (MOCs), and run-time systems. In this section, we describe and evaluate these alternatives.

The *library of application components approach* exports a collection of manually designed blocks to the application programmer, who stitches these together to create his application. The advantage of such an approach is a better mapping to the underlying hardware, since the components are hand-coded. In addition, these components implement an abstraction that is natural for an application writer as the components are often similar to application model primitives. The disadvantage of this approach is the need to implement every element of the library by hand. If only a limited number of library elements are needed, this approach may be successful. However, in practice, we suspect a large number of elements are needed as application diversity grows. This problem is further compounded when a number of variants of each library element is needed [8].

A *programming languages approach* utilizes a programming language that can be compiled to the target architecture. With this approach, a compiler needs to be written only once for the target architecture and all compiler optimizations can be applied to all applications that are written for the architecture. The principal difficulty with this approach is the requirement to compile to heterogeneous architectures with multiple

processors, special purpose hardware, numerous task-specific memories, and various buses. In addition, the programming abstraction required to effectively create a compiler for such architectures would likely force the programming language to include many architectural concepts that would be unnatural for the application programmer. Examples of this alternative include the numerous projects that have altered the C programming language by exposing architectural features [7] [9].

Another class of approaches uses *refinement from formal models of computation* (MOCs) to implement applications. Models of computation define formal semantics for communication and concurrency. Examples of common MOCs include Kahn Process Networks [10], synchronous dataflow [11], and discrete-event. Because they require applications to be described in an MOC, these approaches are able to prove properties of the application (such as maximum queue sizes required and static schedules that satisfy timing constraints). This class of solutions also emphasizes application modeling and simulation [12]. The disadvantage of this method is that implementation on heterogeneous architectures is inefficient because most implementation paths require significant compiler support. As an example, Edwards has written a compiler to implement designs described in Esterel, a language that implements the synchronous/reactive MOC [13]. However, his work generates C code and relies on a C compiler for implementation on target architectures [14]. In addition, the MOCs used by these approaches may not be natural design entry environments. For example POLIS requires all applications to be expressed in co-design finite state machines [12].

*Run-time systems* offer another category of solu-

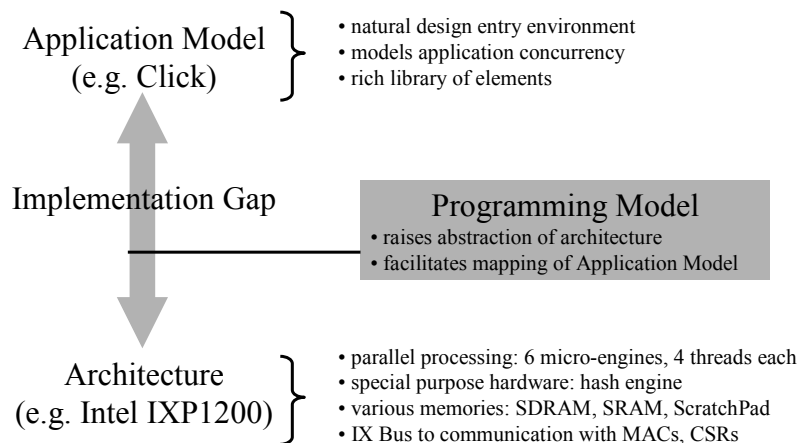


Figure 2. Implementation Gap.

tions to the *implementation gap*. Run-time systems introduce dynamic operation (e.g. thread scheduling) that enables additional freedom in implementation. This also can be used to present the programmer with an abstraction of the underlying architecture (e.g. a view of infinite resources). While run-time systems are necessary for general-purpose computation, for many data-oriented embedded applications (like data plane processing) they introduce additional overhead at run-time. Additionally, some ASIP architectures have included hardware constructs to subsume simple run-time system tasks like thread scheduling on the IXP1200 and inter-processor communication (ring buffers on the Intel IXP2800 [15]). Examples of this approach include VxWorks [16] and the programming interface for the Broadcom Calisto [17].

Based on the trade-offs between the above approaches, we propose a programming model that is a hybrid of the application component library and programming language approaches. We describe our approach in the next section.

## 4. NP-Click: A Programming Model for the Intel IXP1200

In this section, we describe NP-Click, our programming model as implemented on the Intel IXP1200. NP-Click combines an efficient abstraction of the target architecture with features of a domain-specific language for networking. The result is a natural abstraction that enables programmers to quickly write efficient code. The model is designed to ease three major difficulties of programming network processors: taking advantage of hardware parallelism, arbitration of shared resources, and efficient data layout. This section describes the main components of the programming model: elements and their communication, threading, and arbitration of shared resources. We also give hints for using NP-Click to arrive at an efficient implementation.

### 4.1. Overview of the Model

Our programming model integrates concepts from Click to provide a natural design entry environment, and an abstraction of the IXP1200 architecture in order to leverage the computational power of the device.

To describe applications, we borrow Click's simple yet powerful abstraction of elements communicating by passing packets via push and pull semantics. Since our initial studies of the IXP1200 architecture showed the importance of multi-threading to hide memory and communication latency, we chose to export thread boundaries directly to the application programmer.

Unlike Click's implementation, elements in our programming model are implemented in IXP-C, the subset of C the IXP1200 supports. In addition, due to the performance impact of data layout on the target architecture (between registers, Scratchpad, SRAM, and SDRAM), our implementation enables the programmer to effectively use these memories. Since the IXP1200 has separate program memories for each microengine, we allow multiple implementations of the same element of a design to exploit additional application concurrency. However, since most data memory is shared among microengines, the programmer must specify which data is shared among these instances and which data can be duplicated. We also provide the programmer with a machine abstraction API that hides pitfalls of the architecture and exports a more natural abstraction for unique memory features and co-processors. In addition, we provide a library that abstracts shared resources to separate the problem of arbitration from computation.

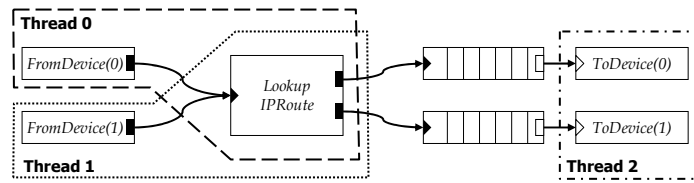
### 4.2. Elements

Computation in our programming model is described in a fashion similar to Click, with modular blocks, called elements, which are sequential blocks of code that generally encapsulate particular packet processing functions. However, in our model, elements are defined using IXP-C, keywords for memory layout, and a machine abstraction API that provides key abstractions of low-level architectural details of the IXP1200.

Before describing the details of our programming model, it is important to understand the distinction between *elements*, *types*, and *instances*. An *element* is a defined functional block within a design that has a *type*, which defines its functionality and the semantics of its ports. There may be multiple elements of the same type in a design. An *instance* is an implementation of an element. Depending on an application's mapping on to the target architecture, an element may have multiple instances to exploit parallelism.

Figure 3 shows a small Click network that illustrates the difference between a type, element, and instance. The boxes in the diagram represent elements. *FromDevice(0)* and *FromDevice(1)* are multiple elements of the same type. *LookupIPRoute* is a single element with multiple instances (i.e. it is implemented by Thread 0 and Thread 1).

**4.2.1. Data Layout.** As a significant portion of implementation speed is due to memory access latency, we provide some mechanisms when describing an element to guide memory layout.



**Figure 3. Example Packet Forwarder.**

To separate memory layout concerns from computation, we supply a facility to describe data by its scope. We provide four data descriptors:

- **Universal:** data that is shared among all types
- **Global:** data that is shared among all elements of a specific type
- **Regional:** data that is shared among all instances of a specific element
- **Local:** data that is local to an instance

The *universal* data descriptor describes data that needs to be accessible by all elements. Since this descriptor breaks the element abstraction, we aim to minimize the use of this descriptor. To date, we have not encountered applications that require this construct. We suspect it will mostly be used as an optimization.

*Global* data descriptors are used for data that must be shared across all elements of a given type. It could be used for a shared hardware resource that all elements of a particular type must use. For example, metering elements need to access a clock to determine the rate of packet flow.

Since elements in a Click design may be instantiated multiple times for performance reasons, the *regional* type modifier describes data within an element that must be shared across instantiations. For example, a *LookupIPRoute* element, which looks up the destination port of a packet, requires a large amount of storage for the routing table. As a result, to implement multiple threads that contain instances of the *LookupIPRoute* element without duplicating the entire routing table in memory, as shown in Figure 3, the lookup table must be shared among them, but not amongst different *LookupIPRoute* elements.

The *local* data descriptor is used for state local to an element that need not be shared across multiple instantiations of an element. Examples of this type include temporary variables and loop counters.

Our abstraction is built on top of the *declspec* construct used in IXP-C to bind data to a particular memory (e.g. SRAM, SDRAM, Scratchpad) at compile time. This may be used by the programmer for additional visibility into the memory architecture to im-

prove performance of the implementation by specifying, for example, that certain large data structures, like routing tables, be placed in a large memory.

**4.2.2. Machine API.** In addition to data descriptors, our programming model hides some of the nuances of the Intel IXP1200 architecture. These abstractions are used in conjunction with IXP-C to describe computation within an element.

The IXP1200 implements special purpose hardware for tasks that are commonly executed in software. To shield the programmer from the details of interacting with these hardware blocks, we export an application-level abstraction that encapsulates common uses of the hardware. For example, the IXP1200 has 8 LIFO (“last in, first out”) registers that implement the common stack operations (*push* and *pop*) in a single atomic operation. However, these operations do not, for example, perform bounds checking or thread safety checks. We implement a lightweight memory management system that exposes a natural interface, namely *malloc()* and *free()* which makes use of the LIFO registers to implement a thread-safe freelist that performs bounds checking. These abstractions enable the programmer to reap the performance advantage of special-purpose hardware without understanding particulars of their implementation.

### 4.3. Communication

Our programming model borrows the communication abstraction from Click [4]: elements communicate only by passing packets with push or pull semantics. However, our implementation of this abstraction is quite different from that of the Click software.

We define a common packet data layout that all elements use. We use a packet descriptor, allocated to SRAM, which stores the destination port and the size of the packet. The packet itself is stored in SDRAM. We define methods for reading and writing packet header fields and packet bodies, so these implementation details are hidden from the user.

As an optimization, we implement the packet communication by function calls that pass a pointer to the

packet descriptor and not the packet itself. We enforce that compute elements not send the same packet to multiple output ports to ensure that only one element is processing a particular packet at any given time. The packet data layout provides an abstraction that efficiently communicates packets among elements, but shields the programmer from the specifics of the IXP1200's memory architecture.

#### 4.4. Threading

Arriving at the right allocation of elements to threads is another key aspect to achieving higher performance. Thus, we enable the programmer to easily explore different mappings of elements to threads on the IXP1200. While we believe this task may be automated in the future, given the great performance impact of properly utilizing threads, we make thread boundaries visible to the programmer.

As observed in [5], paths of push (pull) elements can be executed in the same thread by simply calling the source (sink). We implement a similar mechanism, however, because of the fixed number of threads on the IXP1200, we also allow the programmer to map multiple paths to a single thread. To implement this, we synthesize a scheduler that fires each path within that thread. For example, to implement the design in Figure 3, we would synthesize a round-robin scheduler for the schedulable elements in Thread 2 (*ToDevice(0)* and *ToDevice(1)*). We hide the details of how to implement multiple schedulable elements within a thread from the user, but still give them the power to define thread boundaries at the element level.

#### 4.5. Arbitration of Shared Resources

The amount of parallelism present in the target architecture places pressure on shared resources. For example, the IXP1200 has 24 threads that may each simultaneously request a control status register. Such situations lead to potential hazards that necessitate arbitration schemes for sharing resources. To recognize the importance of sharing common resources, we separate arbitration schemes from computation and present them as interfaces to the elements. The two main resources that require arbitration on the IXP1200 are control status registers and the Transmit FIFO.

Control status registers are used to communicate with the MACs (e.g. determining which ingress ports have new data, which egress ports have space). Our experiments have shown access times to the control status registers ranging from 9 to >200 clock cycles, with multiple simultaneous accesses sometimes leading to deadlock. The variability is due to contention on a

common bus (the command bus) used for issuing SDRAM, SRAM, Scratchpad, IX Bus, and control status register commands. This bus quickly saturates with multiple threads checking the status of the MAC at the same time. This variability is a critical factor in determining performance and one of the major difficulties of programming the IXP1200. To eliminate the need for the programmer to cope with this variability, we implement a per-microengine restriction on the number of concurrent control status register accesses. If a thread attempts to access a control status register while the maximum threshold of accesses are outstanding, a context swap is performed and another thread is loaded. While this may reduce overall microengine computational efficiency, this significantly reduces the variability in control status register access times. This abstraction wraps all reads and writes to the control status registers and is transparent to the programmer. This gives the user enough visibility to interact with peripherals efficiently while not being able to saturate the command bus.

The Transmit FIFO (TFIFO) is the principal mechanism for sending data off-chip that is shared by all threads. It is a 16-entry buffer used to queue data to be sent to the MAC. Each entry in the buffer has a flag that indicates whether that entry contains valid data. Microengines send data to the MAC by loading data into a TFIFO buffer entry and then setting the valid flag. A hardware state machine in the IX Bus Interface steps through the TFIFO as a circular buffer, waiting at each entry for the valid flag to be set. Once the valid flag is set for the current entry, the data in the entry is sent to the MAC and the state machine steps to the next entry. Microengines may query the valid flags and the entry the state machine is currently pointing to. Due to the unique nature of the TFIFO and the numerous threads that may be accessing it, managing the arbitration to this resource is a difficult problem. Perhaps the simplest arbitration scheme is to map *a priori* each port to one entry in the TFIFO. This eliminates the overhead associated with runtime coordination between elements but does not scale to applications with more than 16 ports. In addition, this scheme only performs well on an evenly distributed packet mix as the state machine will wait at a TFIFO entry even if there is no data to be sent on that port, thereby slowing the entire system. A slightly more complicated scheme that avoids these limitations is to allow threads to checkout any entry in the TFIFO. This requires a variable in shared memory to coordinate the free locations and mutex locking and unlocking. While this is able to scale functionally and can better handle a burst of data on a port, a good deal of runtime overhead is needed. To allow different arbitration schemes, we present an

interface of the TFIFO to the elements. The implementation of this interface can be customized by the user without modifying the elements themselves.

#### 4.6. Hints for Efficient Implementation

Modularity is widely accepted as a way of productively writing code while promoting reuse and problem partitioning but often at the cost of lower performance. In this section, we describe some hints for using NP-Click to achieve efficient implementations.

First an element may exist in a thread which has multiple push paths or pull paths to service. To ensure the thread is making progress, elements should yield control when waiting on a long latency activity to complete or an intermittent event to occur. This is a coarser version of swapping threads on a multithreaded processor to hide memory access latency. For example, *ToDevice* often polls the MAC to determine whether to send more data. In this case, it is better for a thread if *ToDevice* checks the MAC once, then (if false) moves to another schedulable element. Whereas a multithreaded processor may implement thread swapping with multiple program counters and a partitioned memory space, swapping at the element level may be performed with static variables in an element instance.

While the programming model as presented may be sufficient for some applications, we concede it will fall short for others. As part of the path to final implementation, we provide facilities to further improve performance. These may include configuration-specific enhancements that might be encapsulated in a single element or optimizations across elements (like a specific scheduler for a set of schedulable elements). The enhancements we have used to date are minor changes to a single element or small modifications of arbitration schemes that greatly improve performance. By using NP-Click, the programmer is able to quickly pinpoint performance bottlenecks in the implementation.

### 5. Results

We explore the effectiveness of our programming model by using it to describe the data plane of an IPv4 router and implementing this application on an Intel IXP1200. This section describes the application we implemented, the experimental setup used to gather data, and our results for maximum data rates for numerous packet mixes.

#### 5.1. Application Description

To test our programming model, we used it to implement the data plane of a 16 port Fast Ethernet IP Version 4 router [18]. This application is based on the network processor benchmark specified in [19]. The major requirements of our application are listed below:

- A packet arriving on port  $P$  is to be examined and forwarded on a different port  $P'$ . The next-hop location that implies  $P'$  is determined through a longest prefix match (LPM) on the IPv4 destination address field. If  $P = P'$ , the packet is flagged and forwarded to the control plane.
- The packet header and payload are checked for validity and packet header fields checksum and TTL are updated.
- Packet queue sizes and buffers can be optimally configured for the network processor architecture unless large buffer sizes interfere with the ability to measure sustained performance.
- The network processor must maintain all non-fixed tables (i.e. tables for route lookup) in memory that can be updated with minimal intrusion to the application.
- Routing tables should be able to address any valid IPv4 destination address and should support up next-hop information for up to 64,000 destinations simultaneously.

Figure 4 shows a graphical representation of the Click description of the router. We allocate 16 threads (4 microengines) for receiving packets and 8 threads (2 microengines) for transmitting packets.

#### 5.2. Testing Procedure

To test our implementation, we used a software architecture simulator of the IXP1200 assuming a micro-engine clock rate of 200MHz and an IX Bus clock rate of 100MHz. Our simulation environment also modeled two 8 port Fast Ethernet MACs (Intel IXF440s) connected to the IX Bus. For each port, the IXF440 has 256-byte internal buffers for both ingress and egress traffic.

For measuring performance, we strived to create a realistic testing environment. We tested the router with a 1000 entry routing table whose entries are chosen at random. The destinations of the input packet streams are randomly distributed evenly across output ports. We tested performance with packet streams composed of a single packet size (64, 128, 256, 512, 1024, 1280, and 1518 bytes) and the IETF Benchmarking Methodology Workgroup mix [20]. We considered the router to be functional at a certain data rate if

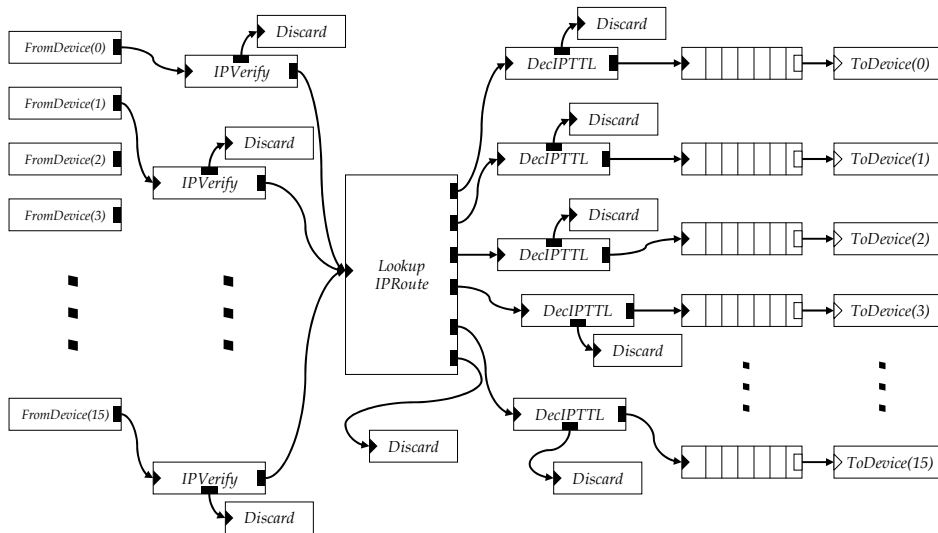


Figure 4. Click representation of IPv4 data plane.

it had a steady-state transmit rate that was within 1% of the receive rate without dropping any packets. We defined *steady state* to be a long interval of time at which data was being constantly received on all ports and data was always ready to be sent on all ports (i.e. no output port was starved).

For each input packet stream, we measured the maximum sustainable data rate. We tested two different implementations of the router, one that includes a hand-coded optimization and one that does not. The optimization was a better arbitration scheme for the Transmit FIFO that did not use shared memory. Our results are shown in Figure 5.

### 5.3. Interpretation of Results

As Figure 5 shows, our packet forwarding implementation without optimizations achieves the same performance regardless of packet size. For this implementation, packet processing is not the bottleneck. Rather, the limitation is a result of the simple arbitration scheme used for the Transmit FIFO. The scheme uses a shared variable among all *ToDevice* elements that requires a locking mechanism to access and update. As a result, the transmit threads spend the majority of their time attempting to acquire this lock. To alleviate this contention, we wrote a configuration-specific arbitration scheme that uses an *a priori* partitioning of the Transmit FIFO which obviates the need for the global variable to be shared across all *ToDevice* elements. Because of NP-Click's separation of computation and arbitration, this hand-coded optimization

was simple to implement as it was fully contained within the Transmit FIFO library so no change to the *ToDevice* element was necessary.

This new scheme results in a much higher maximum data rate across all packet sizes. We believe this illustrates a typical usage of the programming model: NP-Click will be used to quickly gain functional correctness. Performance bottlenecks in the NP-Click implementation are identified, and where needed, hand-coded optimizations will be resorted to. In our experience, these optimizations are easy to implement and localized to an element or a thread.

Comparison of our implementation to published results is difficult because relatively few are available for the Intel IXP1200. Of those results, little information is given about their experimental setup (e.g. IXP1200 and IX Bus clock speed, peripherals used, size of routing table, data rate measurement methodology). These details can have an enormous impact on the reported performance. Hence, for comparison to another implementation, we hand-coded the entire application in IXP-C based on the reference design supplied by Intel [21]. The results of this implementation are also given in Figure 5. The NP-Click implementation (with the arbitration optimization) is able to match the performance of the hand-coded IXP-C implementation for larger packets. For smaller packets, NP-Click is able to achieve 65%-88% of the data rate of the IXP-C implementation. For the IETF Benchmarking Methodology Workgroup packet mix, a representative packet mix, the NP-Click version performs within 7% of the IXP-C version.

The NP-Click version achieves this performance by using a more natural abstraction of the architecture that greatly reduces design time. We estimate NP-Click reduces design time four-fold versus IXP-C. Moreover, NP-Click provides a fast path to initial implementation and produces code that is easier to debug, improve, maintain, and reuse.

We believe our programming model is effective for implementing applications on the IXP1200 and trying different functional partitions across microengines. The modularity and architectural abstraction, however, is responsible for some performance overhead. In particular, the execution time spent in receiving packets and header processing is worse in the NP-Click version due to redundant memory accesses computation. Given the initial results, we are confident we will be able to further close the performance gap by focusing on NP-Click's processing of smaller packets.

## 6. Summary and Conclusions

As application complexity increases, the current practice of programming network processors in assembly language or a subset of C will not scale. Ideally, we would like to program network processors with a network application model, like Click. However, the *implementation gap* between Click and network processor architectures prevents this. In this paper, we define NP-Click, a *programming model* that bridges this gap by coupling the natural expressiveness of Click with an abstraction of the target architecture that enables efficient implementation.

Initial experiments show our programming model

greatly reduces the development time to implement a networking application on the Intel IXP1200 versus current practices. Additional performance tuning is also made significantly easier due to the modularity of the design and the visibility into relevant architectural details.

We used this programming model to implement the data plane of an IPv4 router. Initial results indicate this approach is quite promising, achieving 93% of the performance of a hand-coded 16 port packet forwarder with the IETF Benchmarking Methodology Workgroup packet mix, a realistic set of Internet traffic. Though our approach currently incurs some performance overhead when compared to a hand-coded implementation, we believe we can close this gap in the near future.

As a result, we believe our programming model combines application developer productivity with efficient implementation, which results in a powerful paradigm for programming network processors.

## 7. Future Work

We aim to generalize and improve this work in a number of ways. The first is to quantify the trade-offs in productivity and quality of results for different programming models. We plan to compare our work to other approaches to programming network processors, including assembler, Teja Technologies' Teja NP, and Consystant's StrataNP. This will give us a relative measure of effectiveness for a variety of programming paradigms. A challenge with this project will be properly quantifying productivity.

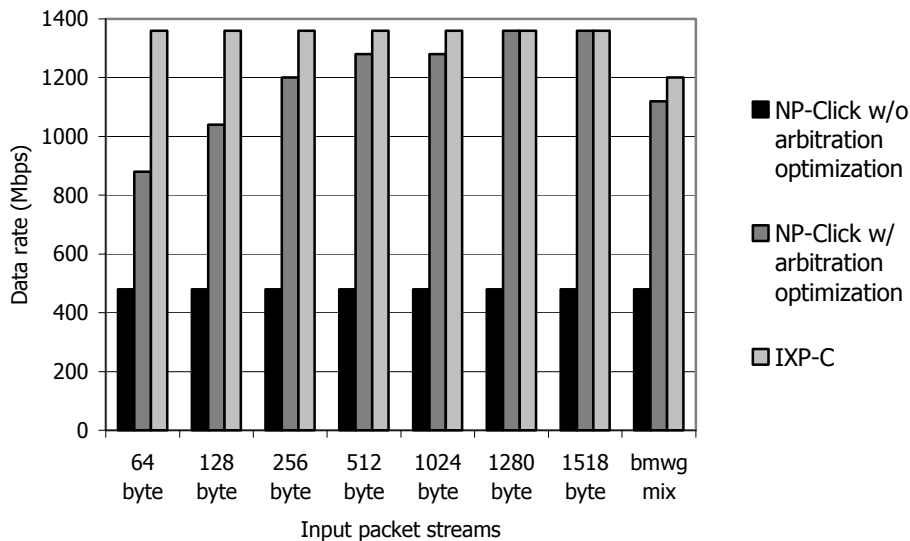


Figure 5. IPv4 Packet Forwarding Results.

The second direction is to implement additional networking applications with our programming model. We are currently considering network address translation (NAT), label edge router/label switch router for multi-protocol label switching (MPLS), and a quality of service (QoS) application.

We also plan to further ease the pain of programming network processors through automation. We aim to provide the programmer with additional automation in the programming flow. Tools could be written to determine thread boundaries, layout data, synthesize scheduling schemes, and perform optimizations based on network topology, similar to optimizations presented in [22].

Lastly, we seek to broaden this work by applying NP-Click to other network processors like the latest Intel IXP architectures (2400, 28xx) and Motorola's DCP C-5. We expect this to be relatively easy for the new IXP architectures, as many of the architectural abstractions can be ported. Generalizing this work to other network processors will have its challenges, but we believe we have identified the key problems in programming network processors: parallelism, arbitration of shared resources, and data layout. The IXP1200 is representative of network processor architectures as it demonstrates a majority of the common features: multiple processing elements, hardware multi-threading, special-purpose hardware, exposed memory regions, and use of peripherals.

## 8. Acknowledgements

This work was supported, in part, by a grant from the Micro-electronics Advanced Research Consortium (MARCO). The authors would like to thank Eddie Kohler, Scott Weber, Andrew Mihal, and the rest of the MESCAL team for their helping in developing NP-Click. We would also like to thank the anonymous reviewers for their feedback on the paper.

## 9. References

- [1] N. Shah. Understanding Network Processors. *Master's Thesis*, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley. 2001.
- [2] N. Shah and K. Keutzer, "Network Processors: Origin of Species," *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*, 2002.
- [3] C. Matsumoto, "Net processors face programming trade-offs," *EE Times*, November 5, 2002, available at <http://www.eetimes.com/story/OEG20020830S0061>.
- [4] E. Kohler et al. The Click Modular Router. *ACM Transactions on Computer Systems*. 18(3), pg. 263-297, August 2000.
- [5] B. Chen and R. Morris, "Flexible Control of Parallelism in a Multiprocessor PC Router," *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001, pg. 333—346.
- [6] Intel Corp., "Intel IXP1200 Network Processor," Product Datasheet, December 2001.
- [7] Intel Corp., "Intel Microengine C Compiler Support: Reference Manual," March 2002.
- [8] J. L. Pino, S. Ha, E. A. Lee and J. T. Buck, "Software Synthesis for DSP Using Ptolemy," *Journal on VLSI Signal Processing*, vol. 9, no. 1, pp. 7-21, January 1995.
- [9] K. Leary, W. Waddington, "DSP/C: A Standard high level language for DSP and Numeric Processing," in *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, 1990, pp. 1065—1068.
- [10] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress 74*. 1974, North-Holland Publishing Co.
- [11] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *IEEE Proceedings*, September 1987.
- [12] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic Publisher, Massachusetts, 1997.
- [13] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, 19(2):87–152, Nov. 1992.
- [14] S. Edwards, "Compiling Esterel into Sequential Code," in *Proceedings of the 37th Design Automation Conference (DAC 2000)*. Los Angeles, California, June 5-9, 2000. pp. 322-327.
- [15] Intel Corp., "Intel IXP2800 Network Processor," Product Brief, 2002.
- [16] Wind River Systems, Inc., "VxWorks Reference Manual," May 1999.
- [17] J. Nickolls et al, "Broadcom Calisto: A Multi-Channel Multi-Service Communications Platform," in *Hot Chips 14*, August 2002.

- [18] F. Baker, "Requirements for IP Version 4 Routers," *Request for Comments - 1812*, Network Working Group, June 1995.
- [19] M. Tsai, C. Kulkarni, C. Sauer, N. Shah, K. Keutzer, "A Benchmarking Methodology for Network Processors", *1st Network Processor Workshop, 8th Int. Symposium on High Performance Architectures*, 2002.
- [20] S. Bradner, J. McQuaid, "A Benchmarking Methodology for Network Interconnect Devices," *Request for Comments - 2544*, Internet Engineering Task Force (IETF), March 1999.
- [21] Intel Corp., "IXP1200 Network Processor Micro-engine C RFC1812 Layer 3 Forwarding Example Design," Application Note, September 2001.
- [22] E. Kohler, R. Morris, and B. Chen, "Programming language optimizations for modular router configurations," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, California, October 2002, pages 251-263.